µC/CAN

CAN Communication Library

User Manual

Version 2.4.1
Manual Rev. 1

Micriμm

Empowering Embedded Systems

www.Micrium.com

Disclaimer

Specifications written in this manual are believed to be accurate, but are not guaranteed to be entirely free of error. Specifications in this manual may be changed for functional or performance improvements without notice. Please make sure your manual is the latest edition. While the information herein is assumed to be accurate, Micrium Technologies Corporation (the distributor) assumes no responsibility for any errors or omissions and makes no warranties. The distributor specifically disclaims any implied warranty of fitness for a particular purpose.

Copyright notice

The latest version of this manual is available as PDF file in the download area of our website at www.micrium.com. You are welcome to copy and distribute the file as well
as the printed version. You may not extract portions of this manual or modify the PDF file in any way without the prior written permission of Micrium Technologie Corporation. The software described in this document is furnished under a license and may only be used or copied in accordance with the terms of such a license.

Trademarks

Names mentioned in this manual may be trademarks of their respective companies. Brand and product names are trademarks or registered trademarks of their respective holders.

Registration

Please register the software via email. This way we can make sure you will receive updates or notifications of updates as soon as they become available. For registration please provide the following information:

- Your full name and the name of your supervisor
- Your company name
- Your job title
- Your email address and telephone number
- Company name and address
- Your company's main phone number
- Your company's web site address
- Name and version of the product

Please send this information to: licensing@micrium.com

Contact address

Micrium Technologies Corporation
1290 Weston Road, Suite 306
Weston, FL 33326
USA
Phone : +1 954 217 2036
FAX : +1 954 217 2037
WEB : www.micrium.com
Email : support@micrium.com

Manual versions

This manual describes the latest software version. The software version number can be found in the table .Software versions. Later in this chapter. If any error occurs, please inform us and we will try to help you as soon as possible. For further information on topics or routines not yet specified, please contact us.

Print date: 2011-11-03

Manual Version	Date	By	Explanation
1.0.0	2005-08-15	MH	Initial Version
1.0.1	2006-10-13	SF	Enhanced resolution of signals and timestamps for signals
1.0.2	2007-01-29	SF	Callback function for signal read and write protection for signals.
1.0.3	2007-03-02	MH	Improve printing quality of figures
1.0.4	2007-03-19	SF	Description of IO-Ctls for driver layer, additional configuration parameter to disable usage of CanBusTx/Rx/NsHandler.
1.0.5	2007-05-25	SF	XXCanInit-funktion parameter arg is used as bus device name.
2.0.0	2008-01-21	SF	Reduced IoCtl-functions in driver. Therefore changes in CAN bus layer. Additional configuration parameter for CAN bus and CAN signal layer. New hook-functions in CAN bus layer. Changed callback-function.
2.0.1	2009-04-08	SF	Changes in CAN bus layer. Description of additional bits in CAN identifier (see Frame Layout).
2.1.0	2009-05-15	MH	Minor changes in API function interfaces.
2.2.0	2009-06-15	SF	Minor changes in CAN config table. (chapter 2.3.2, 4.1.2 and 5.1)
2.2.1	2009-07-06	SF	3.3.2 Description of BusNodeName and DriverDevName.
2.2.2	2009-09-23	SF	4.1.1 Additional information on RX_STANDARD and RX_EXTENDED function codes.
2.3.0	2009-12-01	MH	Describe detailed error codes and adjust the API function return value descriptions. Adjust example with µC/CAN.
2.3.1	2010-01-08	SF	Additional support for µC/OS-III
2.3.2	2010-03-29	SF	Minor changes
2.3.3	2010-03-29	SF	No manual changes
2.4.0	2011-05-10	SF	Rework of internal usage of queues in CanBus-functions.
2.4.1	2011-10-28	SF	Example Correction

Software Version

Software Version	Date	By	Explanation
1.0.0	2005-08-15	MH	Initial Version
1.1.0	2007-03-02	SF	Multiple Rx/Tx queues, timestamps, signal write protection, signal read callback function.
1.2.0	2007-09-04	SF	XXCanInit-arg as dev-ID; additional configuration parameters
2.0.0	2008-01-21	SF	Reduced IoCtl-functions in driver. Therefore changes in CAN bus layer. Additional configuration parameter for CAN bus and CAN signal layer. Added chapter Frame layout / little or big endian.
2.0.1	2009-04-14	SF	Bug Fix in CAN bus layer.
2.1.0	2009-05-15	MH	Improve MISRA compliance in API function interfaces.
2.2.0	2009-06-15	SF	Change of CAN config table to differ between CAN node and driver device.
2.3.0	2009-12-01	MH	Improved error code handling within CAN framework for certification.
2.3.1	2010-01-08	SF	Support for µC/OS-III
2.3.2	2010-03-29	SF	Bug fixes in CanSigDelete and CanMsgDelete
2.3.3	2010-03-29	SF	Bug fixes CanMsgDelete, support for C++, better config-support for µC/CANOpen
2.4.0	2011-05-02	MH	Rework of internal usage of queues.
2.4.1	2011-10-28	SF	Bug Fixes

Table Of Contents

1 Introduction

µC/CAN is a CAN communication library, which simplifies the development of highlevel CAN protocol layers like CANopen, DeviceNET or KWP2000. The library is designed to provide a highlevel interface to CAN communication elements which is configurable and easy to use. Considering strict coding rules during development, the sourcecode is highly efficient in resource usage (in RAM and ROM) and certifiable with a minimum effort.

This guide describes how to configure the µC/CAN communication library. Furthermore the architecture and the using of the library is described step-by-step.

Before reading this guide, you should already have a solid knowledge of the C programming language and the CAN communication. If you feel, that your knowledge in C programming is not sufficient, we recommend the "*The C Programming Language"* by Kernighan and Richie, which describes the programming standard and, in newer editions, also covers the ANSI C standard. Knowledge of assembly programming is not required.

2 µC/CAN Architecture

The architecture of µC/CAN is shown in the following figure:

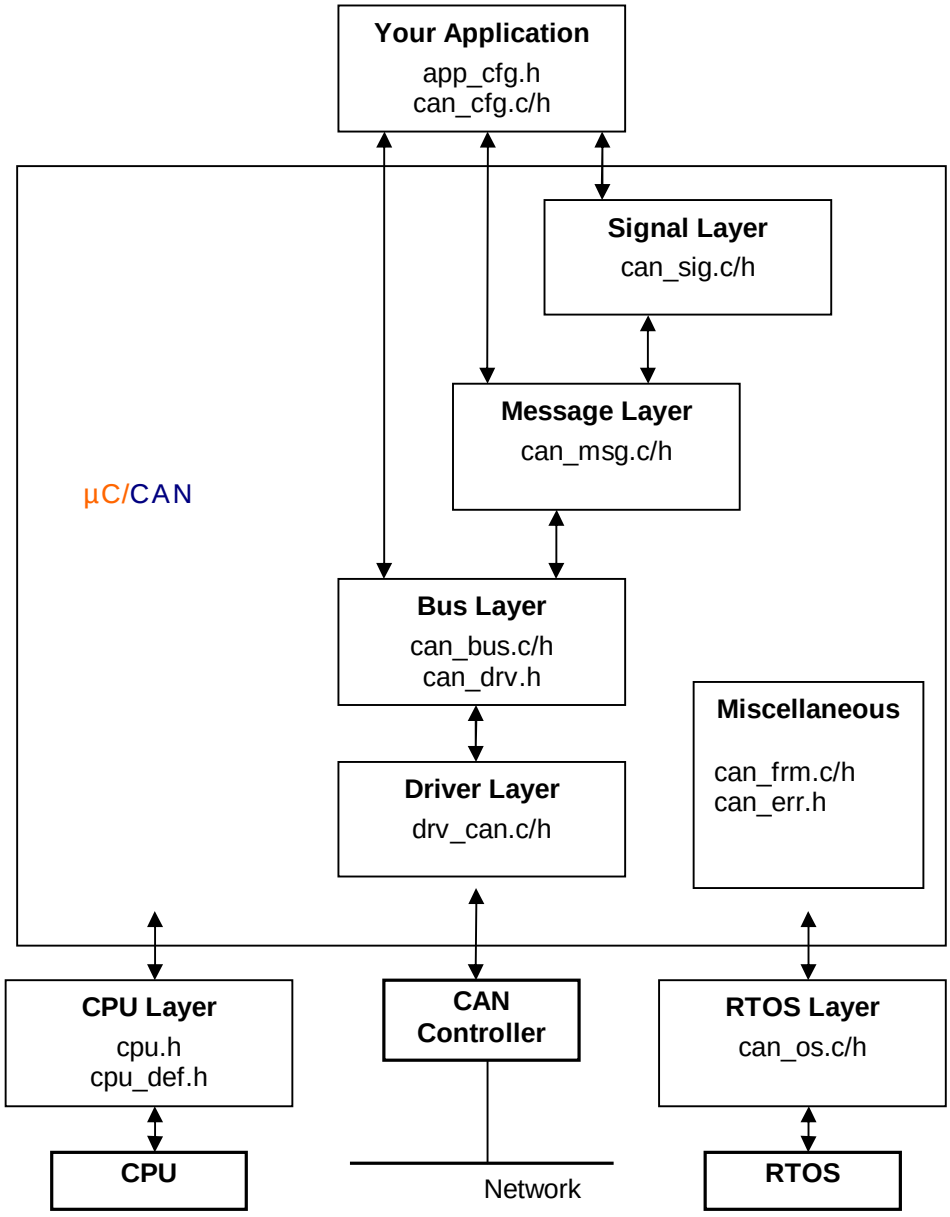

Figure 1-6, Relationship between Application, µC/CAN, CPU, RTOS and CAN-HW

2.1 Signal Layer

This layer is responsible for the CAN signal management. A CAN signal can have a width of 1-32 bit (also selectable as 1,2 or 4byte resolution) and is linked to signal configuration.The signal configuration holds the default status, the default value, the width and a reference to a callback function.

Signals can hold any kind of data, e.g. flags or analog data, etc. Once all messages have been defined, the user can be unconcerned about where the signal is located within a CAN message. The user can act with the CAN signals as they where in a system wide database.

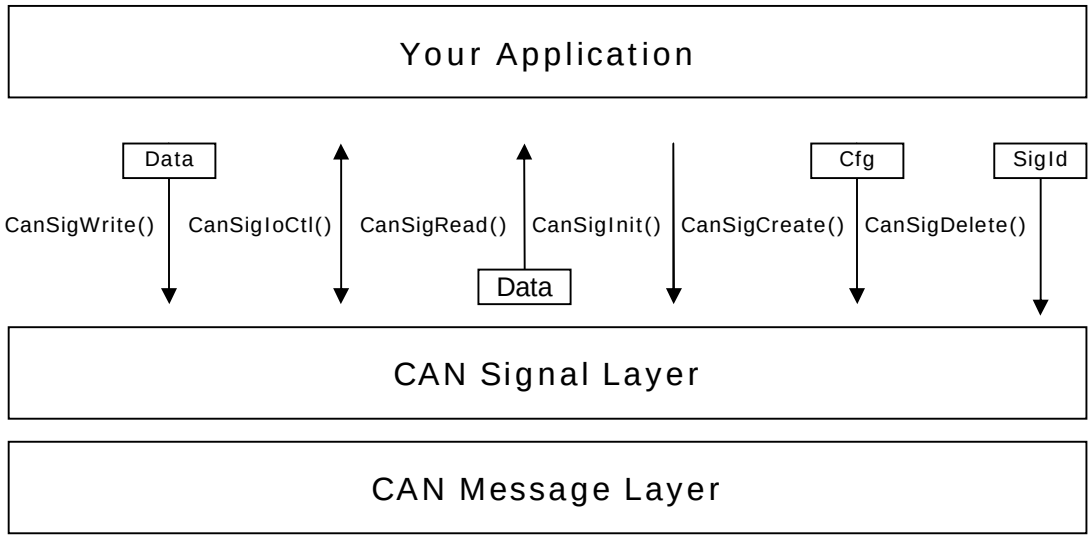

Figure 2-1 : Signal Layer Architecture

The CAN signal layer is configurable during compile time to minimize memory footprint and optimize performance. The following list shows the configuration parameters and their functionality:

Configuration	Meaning	Range	Default
CANSIG_EN	This parameter enables (1) or disables (0) the CAN signal management.	0 / 1	1
CANSIG_N	This parameter defines the maximal number of managed CAN signals.	1 ... 32767	1
CANSIG_ARG_CHK_EN	This parameter enables (1) or disables (0) the argument checking of the CAN bus API functions.	0 / 1	1
CANSIG_MAX_WIDTH	This parameter sets the maximal width of a single integer value in byte.	1, 2 or 4	2

Configuration	Meaning	Range	Default
CANSIG_CALLBACK_EN	This parameter enables (1) or disables (0) the callback function.	0 / 1	1
CANSIG_GRANULARITY	This parameter controls if the width and position of can signal is given in bits or bytes.	BIT / BYTE	BYTE
CANSIG_STATIC_CONFIG	To reduce memory usage, declare a static signal table.	0 / 1	0
CANSIG_USE_DELETE*	To reduce memory usage don't use delete functions for signal.	0 / 1	1

*Note: To use the delete function the configuration parameter CANSIG_STATIC_CONFIG must be set to 0.

2.1.1 Initializing the Signal Database

During the startup phase of the system, the CAN signal database must be initialized. This builds a linked list of unused CAN signal objects.

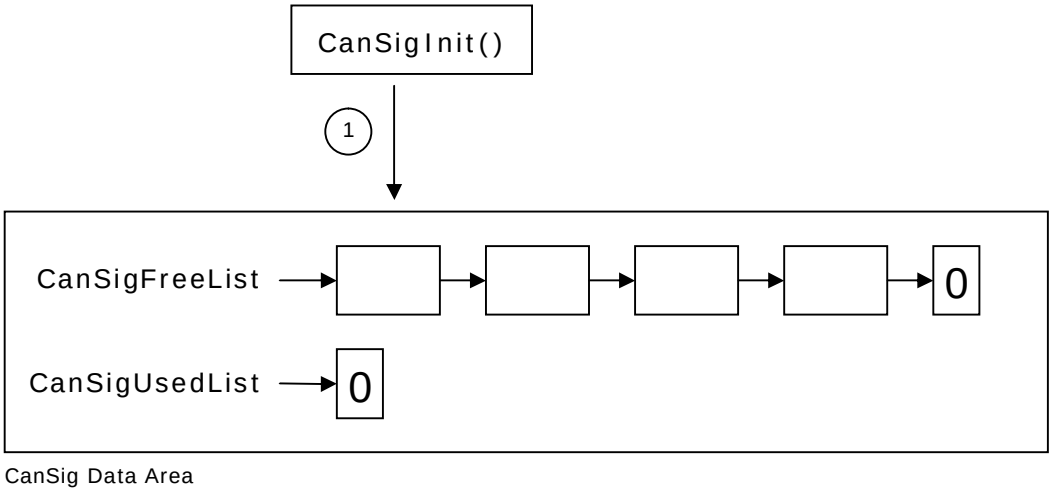

Figure 2-2 : CAN signal database after initialization

1. The function CanSigInit() builds the internal CanSig Data Area. The figure above shows the situation of the CAN signal objects after the initialization.
2. In case of CANSIG_STATIC_CONFIG is enabled, the the signal database is not a linked list of CAN signal objects. Instead it is a simple list where the CAN signal objects are referenced in the order in which the CAN signals are defined. In this case CanSigCreate and CanSigDelete cannot be used.

2.1.2 Creating a CAN Signal

Before the application can use the CAN signal, the CAN signal must be configured with the needed data type parameters.

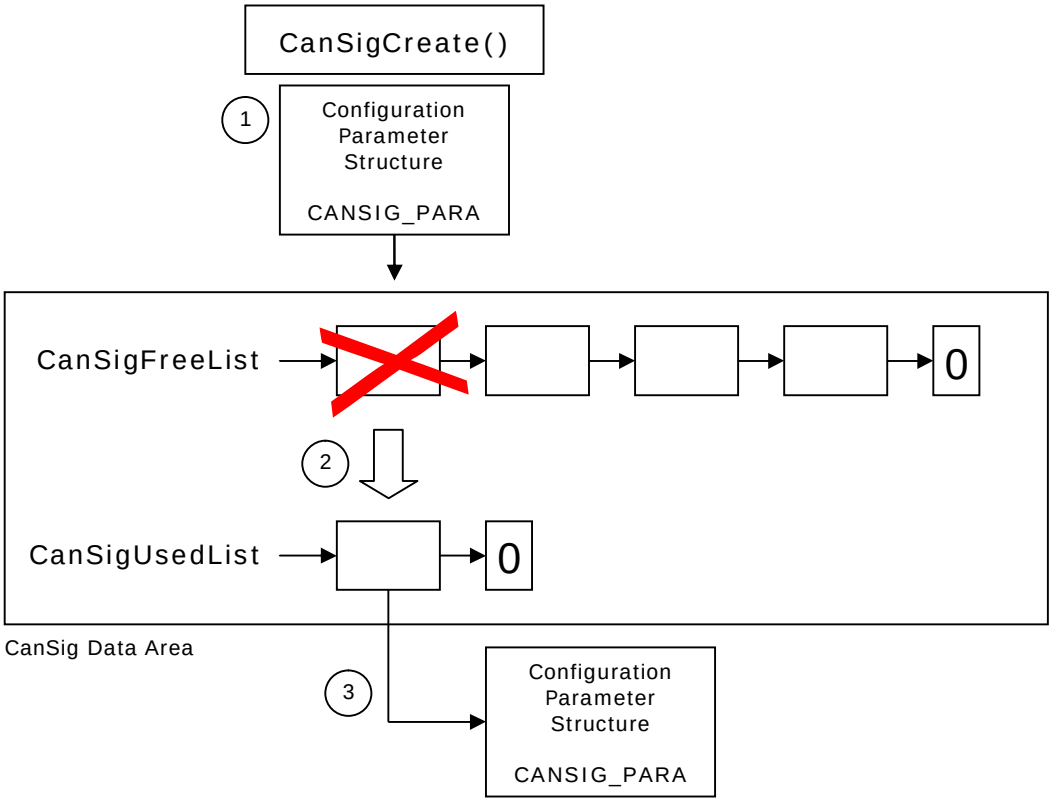

Figure 2-3 : Creating a CAN signal

1. The function CanSigCreate() is called with an application specific CAN signal configuration parameter structure (see documentation of structure CANSIG_PARA).
2. After checking, that a free CAN signal object is available, the object is moved from the free list to the used list.
3. The given CAN signal parameter structure is checked to contain plausible data. If this check was successful, the structure is linked to the CAN signal object.
4. This function cannot be used in case CANSIG_STATIC_CONFIG is enabled.

2.1.3 Deleting a CAN Signal

To reconfigure a CAN signal, the signal must be cleared with the corresponding signal identifier.

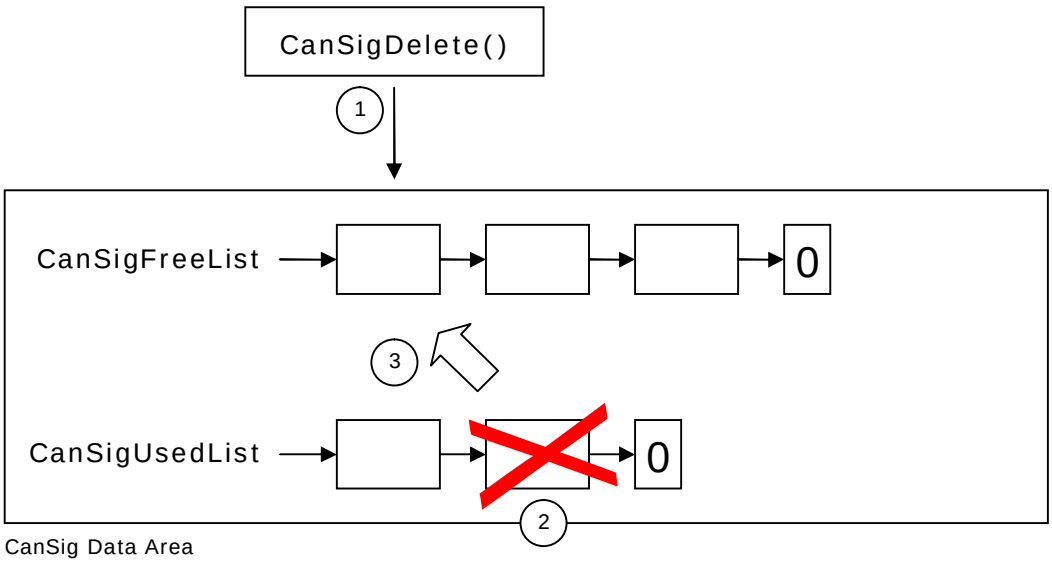

Figure 2-4 : Deleting a CAN signal

1. The function CanSigDelete() is called with the unique CAN signal identifier. This identifier is returned from CanSigCreate() and shall be used to access CAN signal objects.
2. After checking, that the CAN signal object is in use, the signal will be cleared.
3. The signal is moved from the used list to the free list.
4. This function cannot be used in case CANSIG_STATIC_CONFIG is enabled.

2.2 Message Layer

This layer is responsible for the CAN message construction. A collection of CAN signals can be combined to a CAN message. The mapping of the CAN signals can be changed during runtime or configured static.

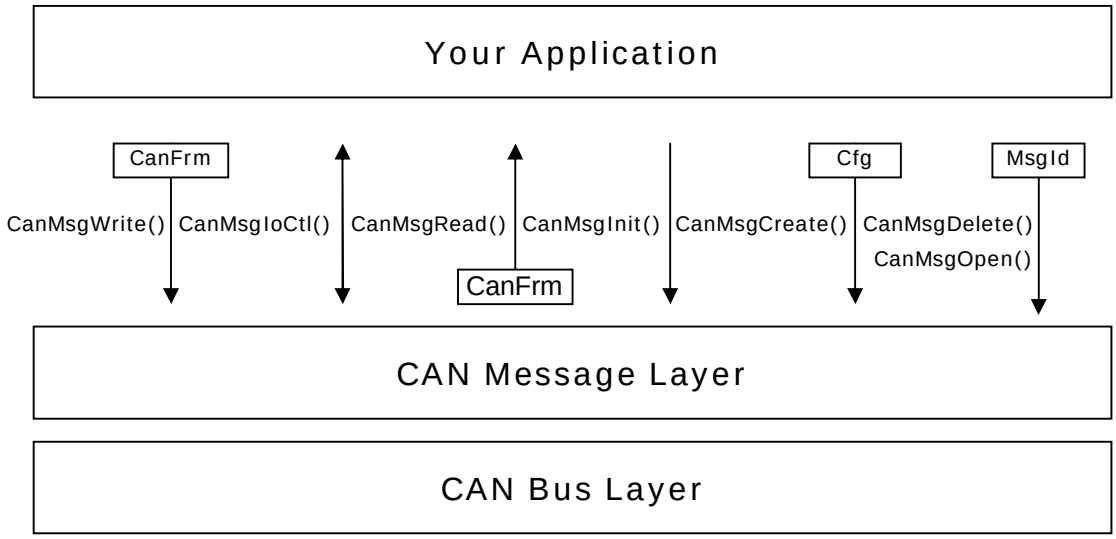

Figure 2-5 : Message Layer Architecture

The CAN message layer is configurable during compile time to minimize memory footprint and optimize performance. The following list shows the configuration parameters and their functionality:

Configuration	Meaning	Range	Default
CANMSG_EN	This parameter enables (1) or disables (0) the CAN message management.	0 / 1	1
CANMSG_N	This parameter defines the maximal number of managed CAN messages.	1 ... 32767	1
CANMSG_ARG_CHK_EN	This parameter enables (1) or disables (0) the argument checking of the CAN bus API functions.	0 / 1	1

2.2.1 Initializing the Message Management

During the startup phase of the system, the CAN message list shall be initialized. This builds a linked list of unused CAN message objects.

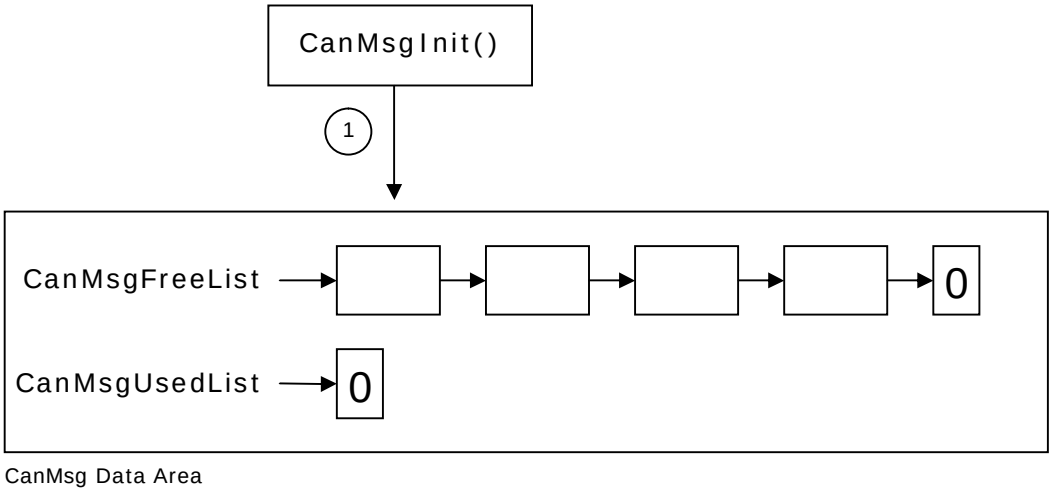

Figure 2-6 : CAN message database after initialization

1. The function CanMsgInit() builds the internal CanMsg Data Area. The figure above shows the situation of the CAN message objects after the initialization.

2.2.2 Creating a CAN Message

Before the application can use a CAN message, the CAN message must be configured with the parameters like CAN Identifier (extended or standard), CAN DLC and the collection of the linked CAN signals.

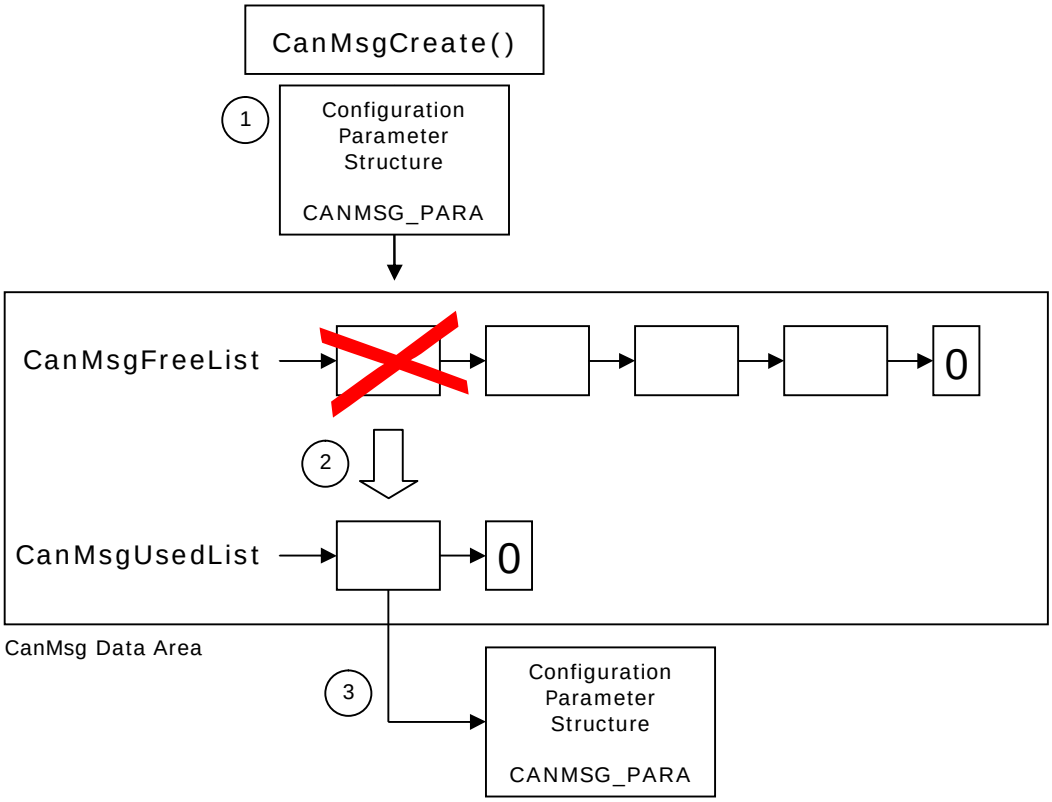

Figure 2-7 : Creating a CAN message

1. The function CanMsgCreate() is called with an application specific CAN message configuration parameter structure (see documentation of structure CANMSG_PARA).
2. After checking, that a free CAN message object is available, the object is moved from the free list to the used list.
3. The given CAN message parameter structure is checked to contain plausible data and all linked CAN signals are in use. If this check was successful, the structure is linked to the CAN message object.

2.2.3 Deleting a CAN Message

If a CAN message is no more used, the message can be cleared with the corresponding message identifier.

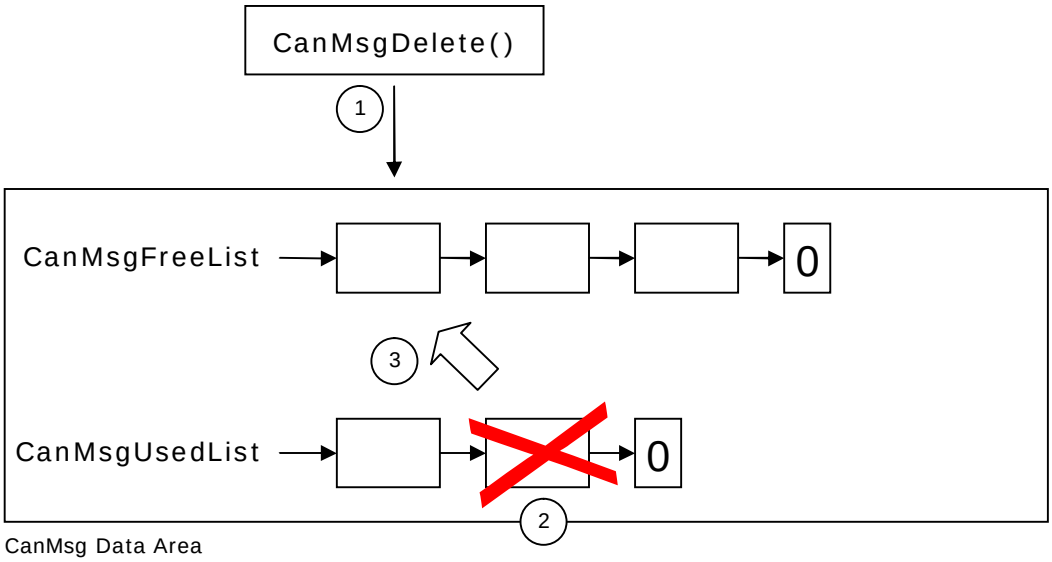

Figure 2-8 : Deleting a CAN message

1. The function CanMsgDelete() is called with the unique CAN message identifier. This identifier is returned from CanMsgCreate() and shall be used to access CAN message objects.
2. After checking, that the CAN message object is in use, the message will be cleared.
3. The message is moved from the used list to the free list.

2.2.4 Opening a CAN Message

To access a CAN message with a known CAN Identifier, this message object can be searched with the opening function.

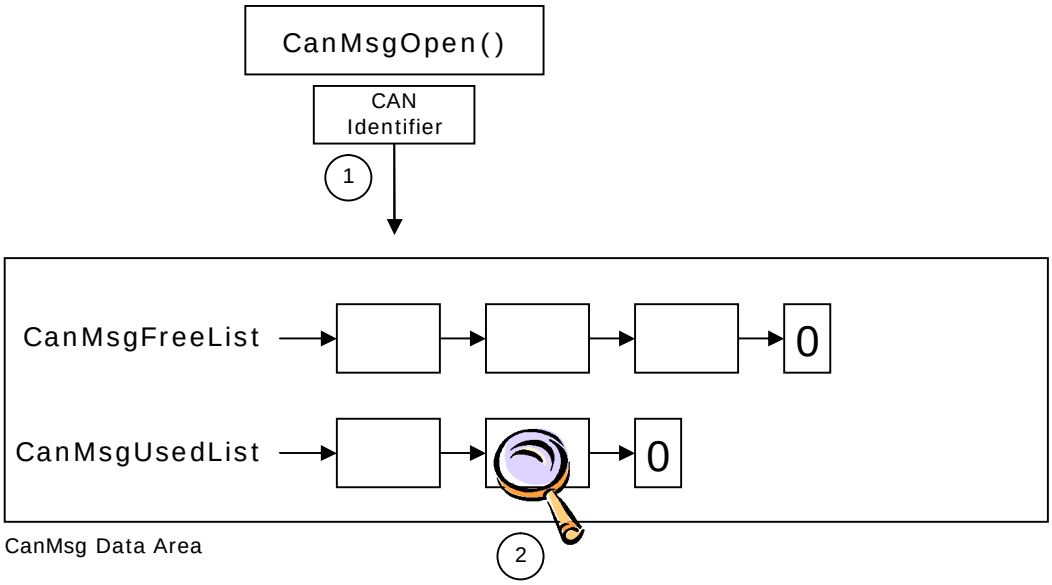

Figure 2-9 : Opening a CAN message

1. The function CanMsgOpen() is called with the CAN Identifier.
2. The function loops through the used list and searches the first message with the given CAN Identifier.

2.2.5 Writing a CAN Message

When a CAN message was received via the CAN bus, the content can be distributed to the linked CAN signals.

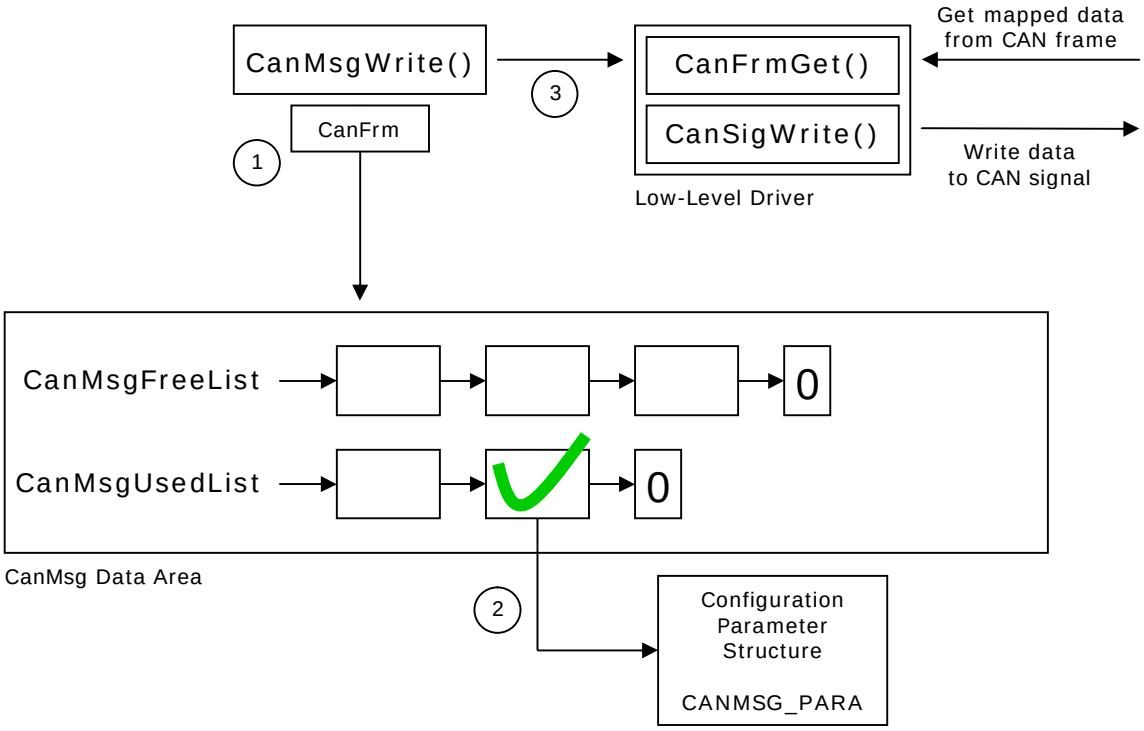

Figure 2-10 : Writing a CAN message

1. The function CanMsgWrite() is called with the unique CAN message identifier and a CAN frame.
2. The function checks, that the given CAN message is in use.
3. The function uses the CanFrmGet() and CanSigWrite() to get the mapped data out of the given CAN frame and write it to the mapped CAN signals.

2.2.6 Reading a CAN Message

When a CAN message shall be transmitted via the CAN bus, the CAN frame can be build with the linked CAN signals.

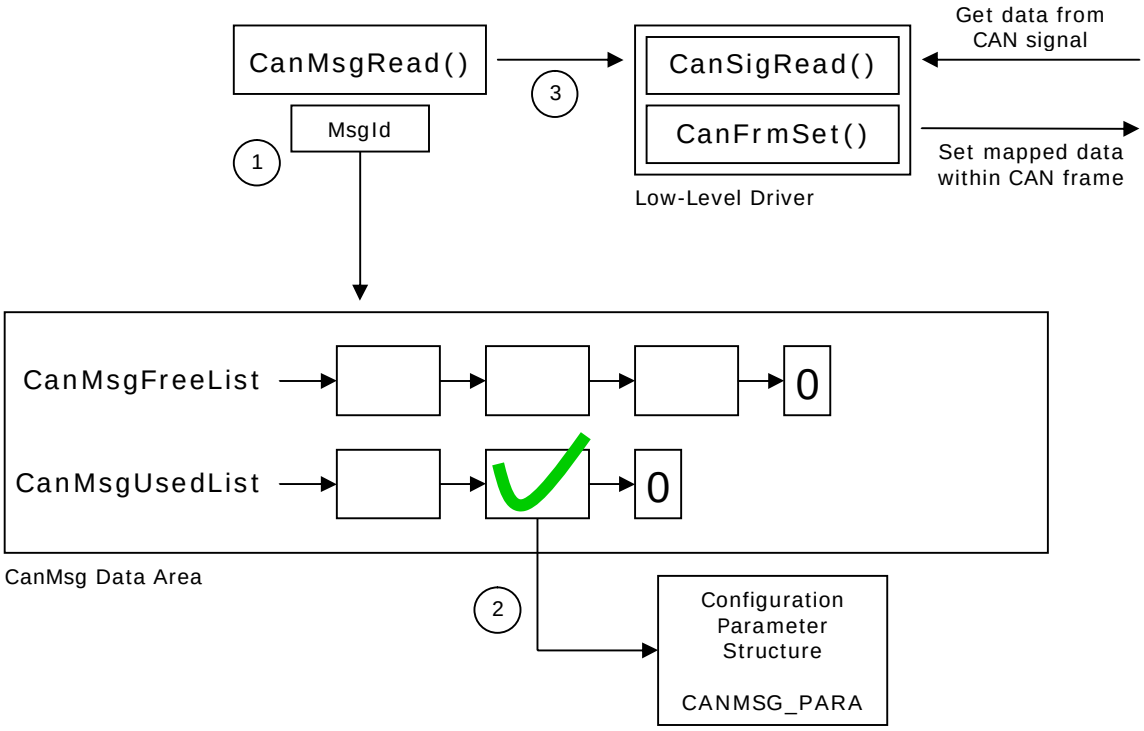

Figure 2-11 : Reading a CAN message

1. The function CanMsgRead() is called with the unique CAN message identifier and a CAN frame.
2. The function checks, that the given CAN message is in use.
3. The function uses the CanSigRead() and CanFrmSet() to get the data from the mapped CAN signals and set the data within the given CAN frame according to the configured mapping information.

2.3 Bus Layer

This layer implements the buffered CAN message handling for one or more busses. All access to the CAN controller shall be done with this layer.

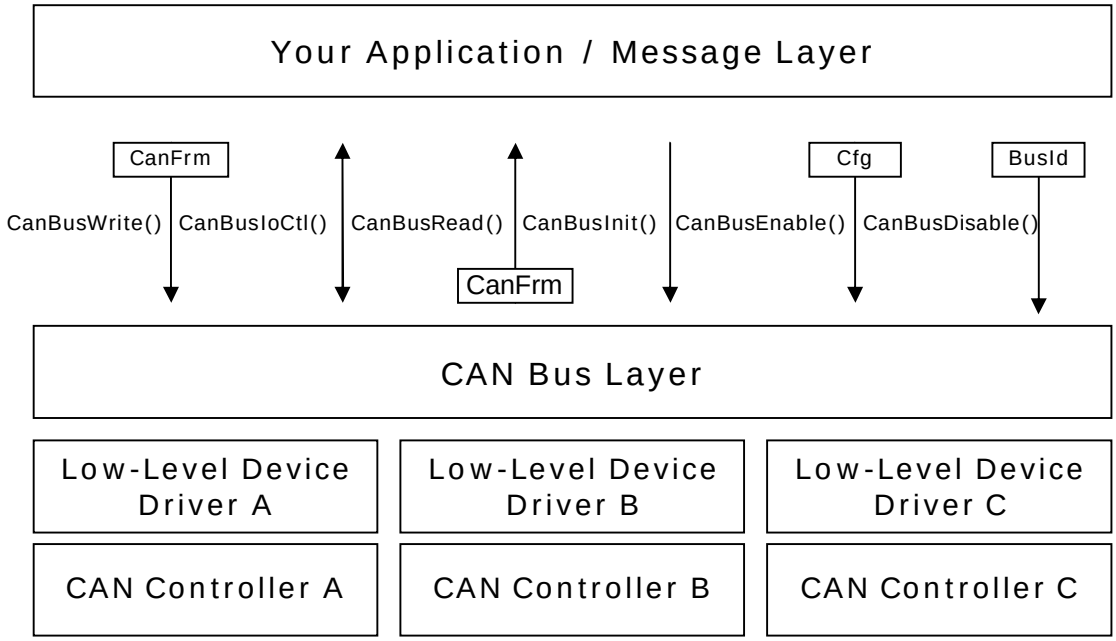

Figure 2-12 : Bus Layer Architecture

The CAN bus layer is configurable during compile time to minimize memory footprint and optimize performance. The following list shows the configuration parameters and their functionality:

Configuration	Meaning	Range	Default
CANBUS_EN	This parameter enables (1) or disables (0) the CAN bus management, e.g. the usage of µC/CAN.	0 / 1	1
CANBUS_N	This parameter defines the maximal number of managed CAN busses.	1 ... 255	1
CANBUS_ARG_CHK_EN	This parameter enables (1) or disables (0) the argument checking of the CAN bus API functions.	0 / 1	1
CANBUS_TX_HANDLER_EN	This parameter enables (1) or disables (0) the usage of function CanBusTxHandler().	0 / 1	1

Configuration	Meaning	Range	Default
CANBUS_RX_HANDLER_EN	This parameter enables (1) or disables (0) the usage of function CanBusRxHandler().	0 / 1	1
CANBUS_NS_HANDLER_EN	This parameter enables (1) or disables (0) the usage of function CanBusNsHandler().	0 / 1	1
CANBUS_STAT_EN	This parameter enables (1) or disables (0) the statistic information collection.	0 / 1	0
CANBUS_RX_QSIZE	The size of the receive queues in CAN frames.	1 ... 65535	2
CANBUS_TX_QSIZE	The size of the transmit queues in CAN frames.	1 ... 65535	2
CANBUS_HOOK_NS_EN	Enable node status handler hook function.	0 / 1	0
CANBUS_HOOK_RX_EN	Enable receive handler hook function.	0 / 1	0

2.3.1 Initializing the CAN Bus Manager

During the startup phase of the system, the CAN bus management must be initialized. This clears all CAN bus objects in the Can bus table.

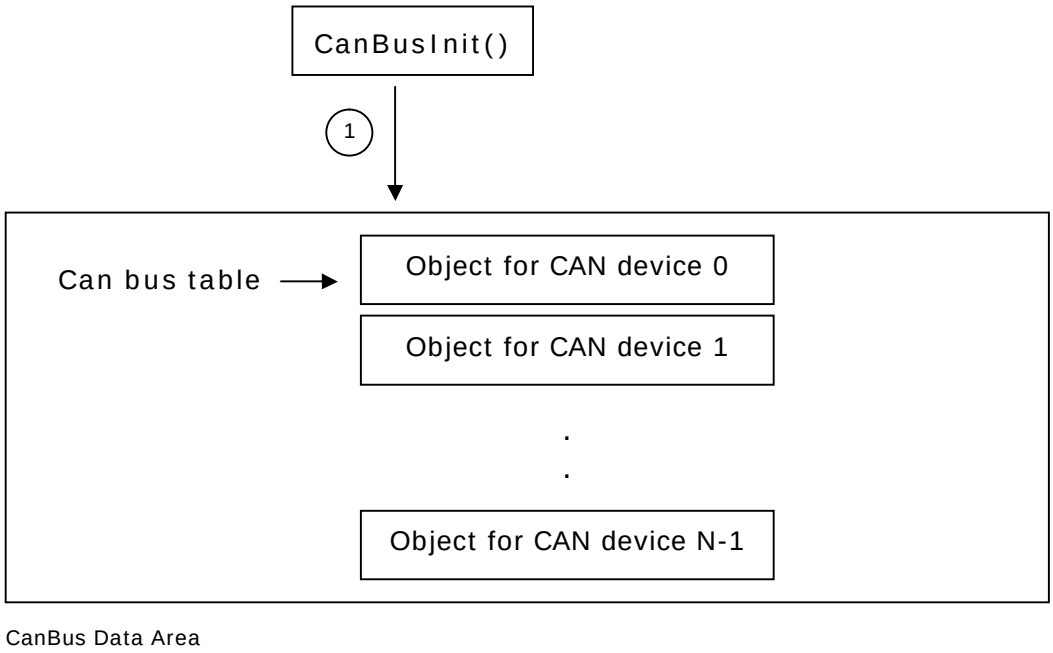

Figure 2-13 : CAN bus objects after initialization

1. The function CanBusInit() creates all needed OS data elements like queues, semaphores, etc.. and builds the internal CanBus Data Area.

Note: there is no access to any CAN controller in the system during this phase.

2.3.2 Enabling the CAN Bus

Before the application have access to the CAN bus, the CAN controller must be configured with the needed communication parameters.

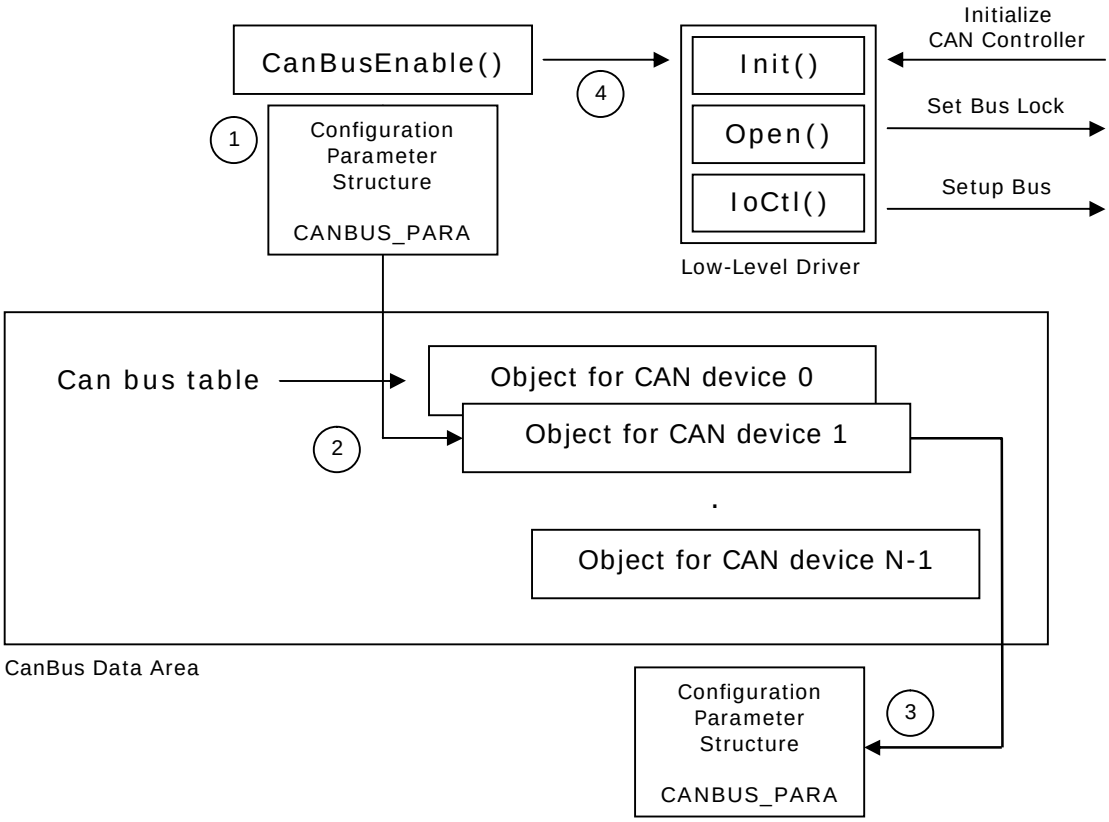

Figure 2-14 : Enabling the CAN bus

1. The function CanBusEnable() is called with an application specific CAN bus configuration parameter structure (see documentation of structure CANBUS_PARA).
2. The bus node in the configuration parameter structure selects the object in the CAN bus table.
3. The given CAN configuration parameter structure is checked to contain plausible data and function pointers to the lowlevel device drivers. If this check was successful, the structure is linked to the CAN bus object.
4. The function pointers to the lowlevel device drivers are used to access the CAN controller. This function uses Init(), Open() and IoCtl() to initialize the CAN controller. In case that the baudrate is configured to 0 then the CAN controller will not be set to active state.

2.3.3 Disabling the CAN Bus

To stop or reconfigure the CAN bus communication, the CAN controller must be stopped with the corresponding bus identifier.

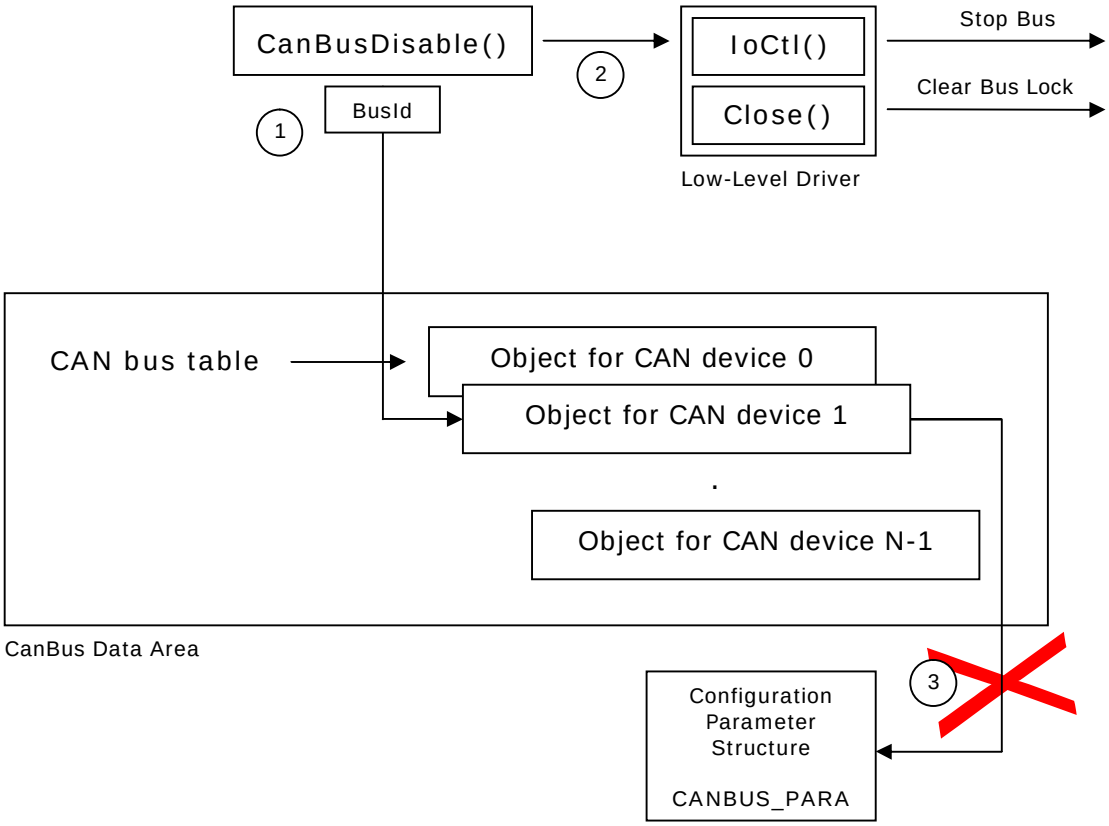

Figure 2-15 : Disabling the CAN bus

1. The function CanBusDisable() is called with the unique CAN bus identifier which selects the object in the CAN bus table
2. After checking, that the CAN bus object is in use, the lowlevel device driver functions IoCtl() and Close() are used to set the bus in passive mode and free the device lock.
3. The object link to the configuration parameter structure is removed.

2.3.4 Transmission while transmitter is idle

If a task or function wants to transmit a CAN frame by calling the function CanBusWrite() while the configured CAN transmitter is idle, the following steps will be performed:

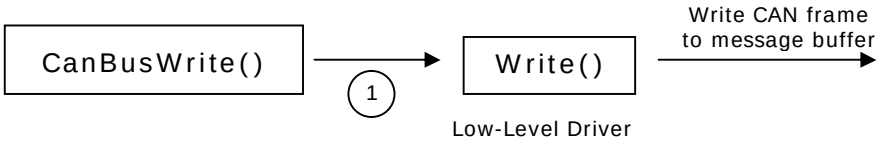

Figure 2-16 : CAN transmission while transmitter is idle

1. The function CanBusWrite() uses the lowlevel device driver function Write() will be used to copy the CAN frame to the CAN controller and enable the transmit complete interrupt.

2.3.5 Transmission while transmitter is busy

If a task or function wants to transmit a CAN frame by calling the function CanBusWrite() while the configured CAN transmitter is busy, the following steps will be performed:

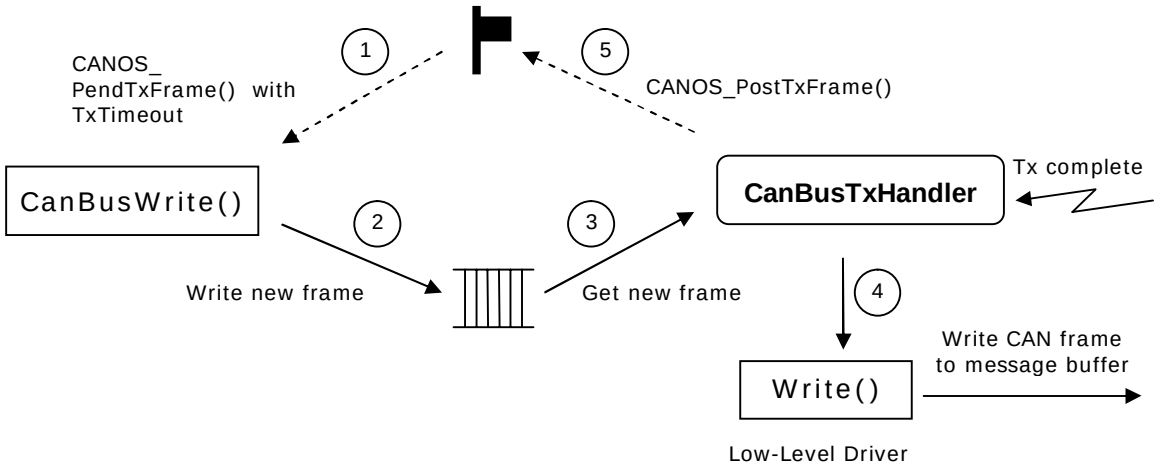

Figure 2-17 : CAN transmission while transmitter is busy

1. The function CanBusWrite() is pending on the counting semaphore with the configured TxTimeout. The timeout must be set with CanBusIoCtl() CANBUS_SET_TX_TIMEOUT
2. If the function has the semaphore within the timeout, the given CAN frame is copied to the internal transmission queue. If a timeout error was detected, the function CanBusWrite() exits with an error-code.

Asynchronous to this actions, the transmission complete interrupt of the CAN controller shall activate the function CanBusTxHandler():

3. The interrupt handler checks if a CAN frame is in the internal transmission queue.
4. If a CAN frame is in the queue, the low level device driver function Write() is used to pass the CAN frame to the CAN controller.
5. The counting semaphore will be posted and the handler will be finished.

2.3.6 Reception of a CAN frame

If a task wants to receive a CAN frame, the function CanBusRead() must be called. The following steps will be performed:

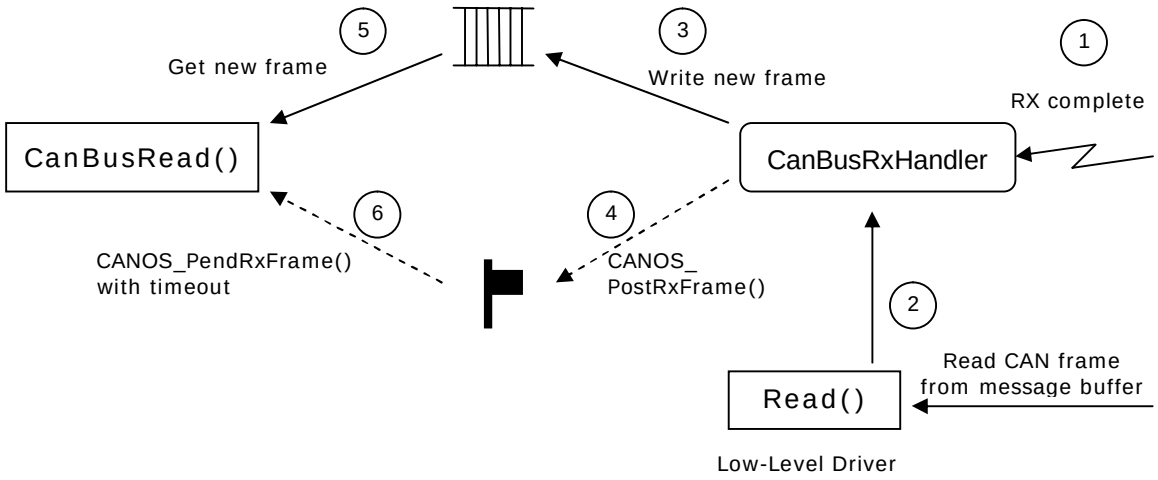

Figure 2-18 : CAN reception

1. With the receive complete interrupt of the CAN controller, the function CanBusRxHandler() shall be activated.
2. The handler uses the low level device driver function Read() to get the CAN frame from CAN controller.
3. The CAN frame will be copied to the internal can receive queue.
4. CANOS_PostRxFrame is called to indicated the new received frame and the handler exits.

Asynchronous to this actions, a task can wait for CAN frames:

5. The function is CanBusRead() pending on the receive semaphore with the configured timeout. The timeout must be set with CanBusIoCtl() CANBUS_SET_RX_TIMEOUT.
6. If a CAN frame is received within the timeout, the content will be copied to the given. In this case the function exists normally. If a timeout is detected, the function exits with an error code.

2.4 CPU Layer

µC/CAN can work with either an 8, 16, 32 or 64-bit CPU but, needs to have information about the CPU you are using. The CPU layer defines such things as the C data type corresponding to 16-bit and 32-bit variables, whether the CPU is little or big endian and, how interrupts are disabled and enabled on the CPU, etc.

CPU specific files are found in the …\uccpu directory and, in order to adapt µC/CAN to a different CPU, you would need to either modify the cpu*.* files or, create new ones based on the ones supplied in the uccpu directory. In general, it´s easier to modify existing files because you have a better chance of not forgetting anything.

2.5 RTOS Layer

µC/CAN assumes the presence of an RTOS but, the RTOS layer allows µC/CAN to be independent of any specific RTOS.

As a minimum, the RTOS you use needs to provide the following services:

- Semaphore
- Timer Services (to get time)

µC/CAN is provided with a µC/OS-II and µC/OS-III interface. If you use a different RTOS, you can use the `can_os.*` for µC/OS-II or µC/OS-III as a template to interface to the RTOS of your choice.

It's also possible to use µC/CAN without an RTOS. In this case the function to get time information needs to be supplied. Note that the supplied file `can_no_os.*` is replacing the OS semaphore and is polling on the flags instead. This might result in a decrease of overall system performance.

3 API Description

This chapter explains all functions of µC/CAN in detail.

3.1 Signal Layer

Description of all data structures, definitions and functions in the signal layer:

Data Structures:

- CANSIG_DATA SIGNAL OBJECT
- CANSIG_PARA SIGNAL CONFIGURATION

Functions:

- CanSigInit() INITIALISE CAN SIGNALS
- CanSigIoCtl() SIGNAL I/O CONTROL
- CanSigWrite() WRITE CAN SIGNAL
- CanSigRead() READ CAN SIGNAL
- CanSigCreate() CREATE CAN SIGNAL
- CanSigDelete() DELETE CAN SIGNAL

3.1.1 CANSIG_DATA

Description:

This structure contains the current status informations for a signal.

Members:

```
typedef struct {
    CPU_INT16U    Id;
    CPU_INT08U    Status;
    CANSIG_VAL_T  Value;
    CPU_INT32U    TimeStamp;
    CANSIG_PARA  *Cfg;
    void         *Next;
} CANSIG_DATA;
```

Member	Meaning
`Id`	The unique signal identifier.
`Status`	The current status of the signal.
`Value`	The current value of the signal. Note: The type of the value 'CANSIG_VAL_T' depends on the configuration setting 'CANSIG_MAX_WIDTH'
`TimeStamp`	Timestamp of last received signal. Note: The signal will be timestamped when a message is written to the signal layer with function CanMsgWrite(..). The time for the timestamp is received from a timer of the underlying OS and therefore has the resolution of the OS timer.
`Cfg`	The pointer to the corresponding CAN signal parameters.
`Next`	The pointer to the next signal in the signal list.

Additional Information:

Status	Meaning
`CANSIG_UNUSED`	This define holds the coding for the information': 'signal not used'. This status is the default status, before the CAN signal initialization function is called. After the CAN signal initialization, this status remains in the CAN signal interface slots, which is not configured via the configuration structure.
`CANSIG_UNCHANGED`	This define holds the coding for the information: 'signal is unchanged'. This status will be set, after a signal access with CanSigRead().
`CANSIG_UPDATED`	This define holds the coding for the information: 'signal is updated'. This status will be set, after a signal write access with CanSigWrite() - independent from the signal value.

Status	Meaning
CANSIG_CHANGED	This define holds the coding for the information: 'signal is changed'. This status will be set, when a CanSigWrite() access changes the value of the signal.
CANSIG_ERROR	This define holds the coding for the information: 'signal error'.

3.1.2 CANSIG_PARA

Description:

This structure contains the configuration informations for a signal. A signal represents a piece of information within the application (a single bit, a bitfield, a integer value, etc...).

Members:

```
typedef struct {
    CPU_INT08U       Status;
    CPU_INT08U       Width;
    CANSIG_VAL_T     Value;
    CANSIG_CALLBACK  CallbackFct;
} CANSIG_PARA;
```

Member	Meaning
`Status`	This member holds the initial status of the signal. This status will be copied to the signal status during initialization phase by calling CanSigInit().
`Width`	This member holds the width of the signal in bit in range 1..32 or in byte (1,2, or 4). Note: Interpreting the width of the signal in bit or byte depends on the configuration setting 'CANSIG_GRANULARITY'
`Value`	This member holds the initial value of the signal. This value will be copied to the signal value during initialization phase by calling CanSigInit().
`CallbackFct`	This member holds the function pointer to the callback function, which is called during a signal write or a read event. The function must have the following Prototype: *void Func (void *, CANSIG_VAL_T *, CPU_INT32U);* The first parameter is used as a pointer to the actual signal. The second parameter is a pointer to the actual write-value. In case of a read-event it is set to 0. The third parameter is used to identify the event type (see below).

```
Note: The callback-function can be en/disabled via can_cfg.h-file.
      Interrupts are disabled when called from signal write function.
      Interrupts are enabled when called from signal read function.
```

Additional Information:

Status	Meaning
`CANSIG_UNUSED`	This define holds the coding for the information: 'signal not used'. This status is the default status, before the CAN signal initialization function is called. After the CAN signal initialization, this status remains in the CAN signal interface slots, which is not configured via the configuration structure.

Status	Meaning
CANSIG_UNCHANGED	This define holds the coding for the information: 'signal is unchang'ed'. This status will be set, after a signal access with CanSigRead().
CANSIG_UPDATED	This define holds the coding for the information: 'signal is updat'ed'. This status will be set, after a signal write access with CanSigWrite() - independent from the signal value.
CANSIG_CHANGED	This define holds the coding for the information: 'signal is changed'. This status will be set, when a CanSigWrite() access changes the value of the signal.
CANSIG_ERROR	This define holds the coding for the information: 'signal error'.
CANSIG_PROT_RO	This define holds the coding for the information 'signal write protection'. When set, CanSigWrite-function will have no effect on that signal.
CANSIG_NO_TIMESTAMP	This define holds the coding for the information 'signal timestamp disabled'. When set, timestamp will not be updated for that signal.

Event type Id	Meaning
CANSIG_CALLBACK_READ_ID	The callback-function is called from the signal read function.
CANSIG_CALLBACK_WRITE_ID	The callback-function is called from the signal write function.

3.1.3 CanSigInit()

Description:

This function will initialize the CAN signal data structure in the following way: all data structures will be linked together and the id will be set to the corresponding index in the data list array.

Prototype:

```
CPU_INT16S CanSigInit(CPU_INT32U  arg);
```

Parameter	Meaning
`arg`	Not used

Additional Information:

- This function has the standard device driver interface, decribed in the porting chapter of the user manual. This allows the CAN signal handling via the standard device driver interface.

- This call is mandantory for dynamic signal configuration. It is omitted when the CAN signal data structure ist defined static, e.g. CANSIG_STATIC_CONFIG is set to 1.

Return Value:

The error code CAN_ERR_NONE is returned.

3.1.4 CanSigIoCtl()

Description:

This function performs a special action on the opened device. The function code defines what the caller want to do.

Prototype:

```
CPU_INT16S CanSigIoCtl(CPU_INT16S  sigId,
                       CPU_INT16U  func,
                       void       *argp);
```

Parameter	Meaning
sigId	Unique signal identifier
func	The functioncode
argp	Pointer to argument, specific to the function code

Additional Information:

Function Code	Meaning
CANSIG_GET_WIDTH	Get the argument pointer width (CPU_INT08U *) and set the content to the signal width.
CANSIG_GET_STATUS	Get the argument pointer status (CPU_INT08U *) and set the content to the signal status.
CANSIG_GET_TIMESTAMP	Get the argument pointer timestamp (CPU_INT32U *) and set the content to the signal timestamp.
CANSIG_GET_TIME_SINCE_UPDATE	Get the argument pointer timestamp (CPU_INT32U *) and set the content to the difference between actual time and signal timestamp.
CANSIG_DISABLE_TIMESTAMP	Timestamping can be disabled to improve performance. The argument pointer is unused.
CANSIG_ENABLE_TIMESTAMP	Enables timestamping. The argument pointer is unused.
CANSIG_SET_TIMESTAMP	The timestamp of a signal will be set to the content of the argument pointer (CPU_INT32U*).
CANSIG_SET_WRITE_PROTECTION	Sets a write protections for a dedicated signal. When set, CanSigWrite-function will have no effect on that signal. The argument pointer (CPU_INT08U*) must be set to CANSIG_PROT_RO to enable write protection or 0 to disable write protection.

Function Code	Meaning
`CANSIG_GET_WRITE_PROTECTION`	Get the argument pointer write protection (CPU_INT08U *) and set the content to write protection status (CANSIG_PROT_RO or 0).

Additional Information:

The IoCtl-functions concerning timestamp are omitted if CANSIG_STATIC_CONFIG is set to 1.

Return Value:

One of the following error codes is returned: CAN_ERR_NONE, CAN_ERR_SIGID, CAN_ERR_NULLPTR, CAN_ERR_NULLSIGCFG or CAN_ERR_IOCTRLFUNC.

3.1.5 CanSigWrite()

Description:

Updates a CAN signal after checking, that this signal is in use.

Prototype:

```
CPU_INT16S CanSigWrite(CPU_INT16S  sigId,
                       void       *buffer,
                       CPU_INT16U  size);
```

Parameter	Meaning
sigId	Unique signal identifier
buffer	Pointer to value
size	Storage size of CAN signal.

Additional Information:

The storage size of the signal is given by parameter CANSIG_MAX_WIDTH.

Return Value:

One of the error codes CAN_ERR_CANSIZE, CAN_ERR_SIGID or CAN_ERR_NULLPTR if an error is detected. Otherwise the storage size of CAN signal is returned.

3.1.6 CanSigRead()

Description:

Reads a CAN signal after checking, that this signal is in use.

Prototype:

```
CPU_INT16S CanSigRead(CPU_INT16S  sigId,
                      void       *buffer,
                      CPU_INT16U  size);
```

Parameter	Meaning
`sigId`	Unique signal identifier
`buffer`	Pointer to value
`size`	Storage size of CAN signal

Additional Information:

The storage size of the signal is given by parameter CANSIG_MAX_WIDTH.

Return Value:

One of the error codes CAN_ERR_CANSIZE, CAN_ERR_SIGID or CAN_ERR_NULLPTR if an error is detected. Otherwise the storage size of CAN signal is returned.

3.1.7 CanSigCreate()

Description:

This function checks, if a free CAN signal is available. If so, this signal is set in front of the used list and initialized with the configured signal data.

Prototype:

```
CPU_INT16S CanSigCreate(CANSIG_PARA  *cfg);
```

Parameter	Meaning
cfg	Configuration of CAN signal

Additional Information:

This function is omitted if CANSIG_STATIC_CONFIG is set to 1.

Return Value:

One of the error codes CAN_ERR_NULLPTR or CAN_ERR_SIGCREATE, if an error is detected. Otherwise the signal identifier is returned.

3.1.8 CanSigDelete()

Description:

This function checks, if a CAN signal is in use. If yes, the CAN signal will be removed from the used list and put back to the free list.

Prototype:

```
CPU_INT16S CanSigDelete(CPU_INT16S  sigId);
```

Parameter	Meaning
`sigId`	Unique signal identifier

Additional Information:

This function is omitted if CANSIG_STATIC_CONFIG is set to 1 or CANSIG_USE_DELETE is set to 0.

Return Value:

One of the error codes is returned: CAN_ERR_NONE, CAN_ERR_SIGID or CAN_ERR_NULLPTR.

3.2 Message Layer

Description of all data structures, definitions and functions in the message layer:

Data Structures:

- CANMSG_DATA CAN MESSAGE OBJECT
- CANMSG_PARA MESSAGE CONFIGURATION

Functions:

- CanMsgInit() INITIALISE CAN SIGNALS
- CanMsgOpen() OPEN A CAN MESSAGE
- CanMsgIoCtl() SIGNAL I/O CONTROL
- CanMsgRead() READ CAN SIGNAL
- CanMsgWrite() WRITE CAN SIGNAL
- CanMsgCreate() CREATE CAN SIGNAL
- CanMsgDelete() DELETE CAN SIGNAL
- CanFrmSet() WRITE VALUE INTO CAN FRAME
- CanFrmGet() READ VALUE OUT OF CAN FRAME

3.2.1 CANMSG_DATA

Description:

This structure contains the dynamic information for a CAN message.

Members:

```
typedef struct {
    CPU_INT16U    Id;
    CANMSG_PARA  *Cfg;
    void         *Next;
} CANMSG_DATA;
```

Member	Meaning
Id	The unique message identifier
Cfg	The pointer to the corresponding CAN message parameters
Next	The pointer to the next message in the message list

3.2.2 CANMSG_LNK

Description:

This structure contains the configuration information for a signal. A signal represents a piece of information within the application (a single bit, a bit field, a integer value, etc...).

Members:

```
typedef struct {
    CPU_INT16U  Id;
    CPU_INT08U  Pos;

} CANMSG_LNK;
```

Member	Meaning
`Id`	The unique signal identifier, which shall be linked to the message.
`Pos`	The position of the first bit (range 0..63) or byte (range 1,2 or 4), which is used in the can frame. Note: Interpreting the position in bit or byte depends on the configuration setting 'CANSIG_GRANULARITY'. Note: The two MSBits in position are used to distinguish the internal coding format. If not set BIG ENDIAN format is used. If set to 01b LITTLE_ENDIAN is used. For detailed description of LITTLE or BIG_ENDIAN usage see chapter 3.2.13 CANSIG_GRANULARITY.

3.2.3 CANMSG_PARA

Description:

This structure contains the configuration information for a signal. A signal represents a piece of information within the application (a single bit, a bit field, a integer value, etc...).

Members:

```
typedef struct {
    CPU_INT32U   Identifier;
    CPU_INT08U   Type;
    CPU_INT08U   DLC;
    CPU_INT08U   SigNum;
    CANMSG_LINK  SigLst[CANMSG_MAX_LINK];

} CANMSG_PARA;
```

Member	Meaning
`Identifier`	The identifier of the CAN message. For more information about types of identifiers please refer to chapter 3.2.13 Frame layout.
`Type`	The type of the message
`DLC`	The DLC of the message
`SigNum`	The used number of signals in the following link table
`SigLst`	This array holds the linked signals for this message

Additional Information:

Type	Meaning
`CANMSG_UNUSED`	This define holds the coding for the information: 'message not used'. This status is the default status, before the CAN message initialization function is called. After the CAN message initialization, this status remains in the CAN message interface slots, which is not configured via the configuration structure.
`CANMSG_TX`	This define holds the coding for the information: 'transmit message'. This status will be set, after a signal is created with a corresponding transmit configuration.
`CANMSG_RX`	This define holds the coding for the information: 'receive message'. This status will be set, after a signal is created with a corresponding receive configuration.

3.2.4 CanMsgInit()

Description:

This function will initialize the CAN message data structure in the following way: all data structures will be linked together and the id will be set to the corresponding index in the data list array.

Prototype:

```
CPU_INT16S CanMsgInit(CPU_INT32U  arg);
```

Parameter	Meaning
arg	Not used

Additional Information:

- This function has the standard device driver interface, decribed in the porting chapter of the user manual. This allows the CAN message handling via the standard device driver interface.
- This call is mandatory.

Return Value:

The error code CAN_ERR_NONE is returned.

3.2.5 CanMsgOpen()

Description:

Searches the CAN message with the given CAN identifier in the used list and returns the unique message identifier.

Prototype:

```
CPU_INT16S CanMsgOpen(CPU_INT16S  drvId,
                      CPU_INT32U  devName,
                      CPU_INT16U  mode);
```

Parameter	Meaning
drvId	Not used
devName	The message identifier
mode	Not used

Return Value:

The message identifier for further access or CAN_ERR_NULLMSG if an error occurs.

3.2.6 CanMsgIoCtl()

Description:

This function allows to control special features of a CAN message.

Prototype:

```
CPU_INT16S CanMsgIoCtl(CPU_INT16S  msgId,
                       CPU_INT16U  func,
                       void       *argp);
```

Parameter	Meaning
msgId	Device id, returned by CanMsgOpen()
func	Function Code
argp	Pointer to argument, specific to the function code

Additional Information:

Function Code	Meaning
CANMSG_IS_CHANGED	Checks all linked signals of the message for changed status. If a least on signal indicates changed, set the flag to TRUE of the given argument pointer (CPU_BOOLEAN *)

Return Value:

One of the following error codes is returned: CAN_ERR_NONE, CAN_ERR_IOCTRLFUNC, CAN_ERR_MSGID, CAN_ERR_SIGID or CAN_ERR_NULLPTR.

3.2.7 CanMsgRead()

Description:

This function constructs a CAN frame out of the linked signals. If there are no linked signals (or the signals are not in use), the corresponding bytes will be 0.

Prototype:

```
CPU_INT16S CanMsgRead(CPU_INT16S   msgId,
                      void        *buffer,
                      CPU_INT16U   size);
```

Parameter	Meaning
`msgId`	Unique message identifier
`buffer`	Pointer to CAN frame of type CANFRM
`size`	Size of buffer (must be equal to sizeof(CANFRM))

Return Value:

One of the error codes CAN_ERR_MSGID, CAN_ERR_NULLPTR or CAN_ERR_FRMSIZE if an error is detected. Otherwise the number of bytes of a CAN frame is returned.

3.2.8 CanMsgWrite()

Description:

This function splits a CAN frame into the linked signals.

Prototype:

```
CPU_INT16S CanMsgWrite(CPU_INT16S  paraId,
                       void       *buffer,
                       CPU_INT16U  size);
```

Parameter	Meaning
`msgId`	Unique message identifier
`buffer`	Pointer to CAN frame of type CANFRM
`size`	Size of buffer in bytes (must be equal to sizeof(CANFRM))

Return Value:

One of the error codes CAN_ERR_MSGID, CAN_ERR_NULLPTR or CAN_ERR_FRMSIZE if an error is detected. Otherwise the number of bytes of a CAN frame is returned.

3.2.9 CanMsgCreate()

Description:

This function checks, if a free CAN message is available. If yes, this message is set in front of the used list and initialized with the configured message data.

Prototype:

```
CPU_INT16S CanMsgCreate(CANMSG_PARA  *cfg);
```

Parameter	Meaning
cfg	Configuration of CAN message

Return Value:

One of the following error codes is returned CAN_ERR_MSGCREATE, CAN_ERR_NULLPTR or CAN_ERR_MSGUNUSED, if an error is detected. Otherwise the message identifier is returned.

3.2.10 CanMsgDelete()

Description:

This function checks, if a CAN message is in use. If yes, the CAN message will be removed from the used list and put back to the free list.

Prototype:

```
CPU_INT16S CanMsgDelete(CPU_INT16S  msgId);
```

Parameter	Meaning
`msgId`	Unique message identifier

Return Value:

One of the following error codes is returned: CAN_ERR_NONE, CAN_ERR_MSGID or CAN_ERR_NULLPTR.

3.2.11 CanFrmSet()

Description:

This function copies a given value with a given width into the frame at the given position. If width or position is out of range, no changes in the frame was done.

Prototype:

```
void       CanFrmSet(CANFRM     *frm,
                     CPU_INT32U  value,
                     CPU_INT08U  width,
                     CPU_INT08U  pos);
```

Parameter	Meaning
`frm`	Pointer to CAN frame
`value`	Value to be inserted into CAN frame
`width`	Width of value in bit in range 1..32 or byte (1, 2 or 4). Note: Interpreting width in bit or byte depends on the configuration setting 'CANSIG_GRANULARITY
`pos`	Position of first bit, which is used in the can frame in range 0..63.or of first byte. Note: Interpreting the position in bit or byte depends on the configuration setting 'CANSIG_GRANULARITY'

Additional Information:

Current limitation of this function: with data width of 1,2 or 4 byte the position range depends on the data width:

- width is byte (1): 0..7
- width is word (2): 0..6
- width is long (4): 0..4

3.2.12 CanFrmGet()

Description:

This function returns the value with a given bit width out of the frame at the given position. If width or position is out of range, the return value is 0.

Prototype:

```
CPU_INT16U CanFrmGet(CANFRM     *frm,
                     CPU_INT08U  width,
                     CPU_INT08U  pos);
```

Parameter	Meaning
`frm`	Pointer to CAN frame
`width`	Width of value in bits in bit in range 1..32 or byte (1, 2 or 4). Note: Interpreting width in bit or byte depends on the configuration setting 'CANSIG_GRANULARITY
`pos`	Position of first bit, which is used in the can frame in range 0..63.or of first byte. Note: Interpreting the position in bit or byte depends on the configuration setting 'CANSIG_GRANULARITY'

Additional Information:

Current limitation of this function: with data width of 1,2 or 4 byte the position range depends on the data width:

- width is byte (1): 0..7
- width is word (2): 0..6
- width is long (4): 0..4

Return Value:

Value out of the CAN frame.

3.2.13 Frame layout / little or big endian

A CAN frame consists of three sections:

- Identifier
- Data
- Data length code

An identifier can be of standard or extended type or can be a remote transmission request.

To differentiate the types the following addition to the identifier is implemented:

- bit31: reserved (always 0)
- bit30: marks a remote transmission request (1=rtr, 0=data frame)
- bit29: marks an extended identifier (1=extended, 0=standard)
- bit28-0: the identifier (standard or extended)

Note: The layers don't use of bit 29 and bit 30, but this bits can be useful in callback- or hook-functions.

The functions CanFrmGet/Set support little or big endianess for bit- and byte-granularity. How bytes are ordered in both formats is well defined, therefore this chapter outlines only the formats for BIT-granularity of signals.

As an example for little endian format we have a 32 bit signal which starts at bit 29 in a 8 byte CAN msg:

	7	6	5	4	3	2	1	0
0	7	6	5	4	3	2	1	0
1	15	14	13	12	11	10	9	8
2	23	22	21	20	19	18	17	16
3	31	30	29 lsb	28	27	26	25	24
4	39	38	37	36	35	34	33	32
5	47	46	45	44	43	42	41	40
6	55	54	53	52	51	50	49	48
7	63	63	61	60 msb	59	58	57	56

Figure 3-1 : Frame Layout little endian format

If we set the signal to 0x12345678 an send the CAN message with all other bits set to 0, we will receive a CAN message with 8 data bytes: 00 00 00 00 CF 8A 46 02.

If we look a the last 5 byte in bit-format:

Bytes	3		4		5		6		7	
Hex	0	0	C	F	8	A	4	6	0	2
Bits	0000	0000	1100	1111	1000	1010	0100	0110	0000	0010

The red-colored bits are not part of the signal and can be dismissed.

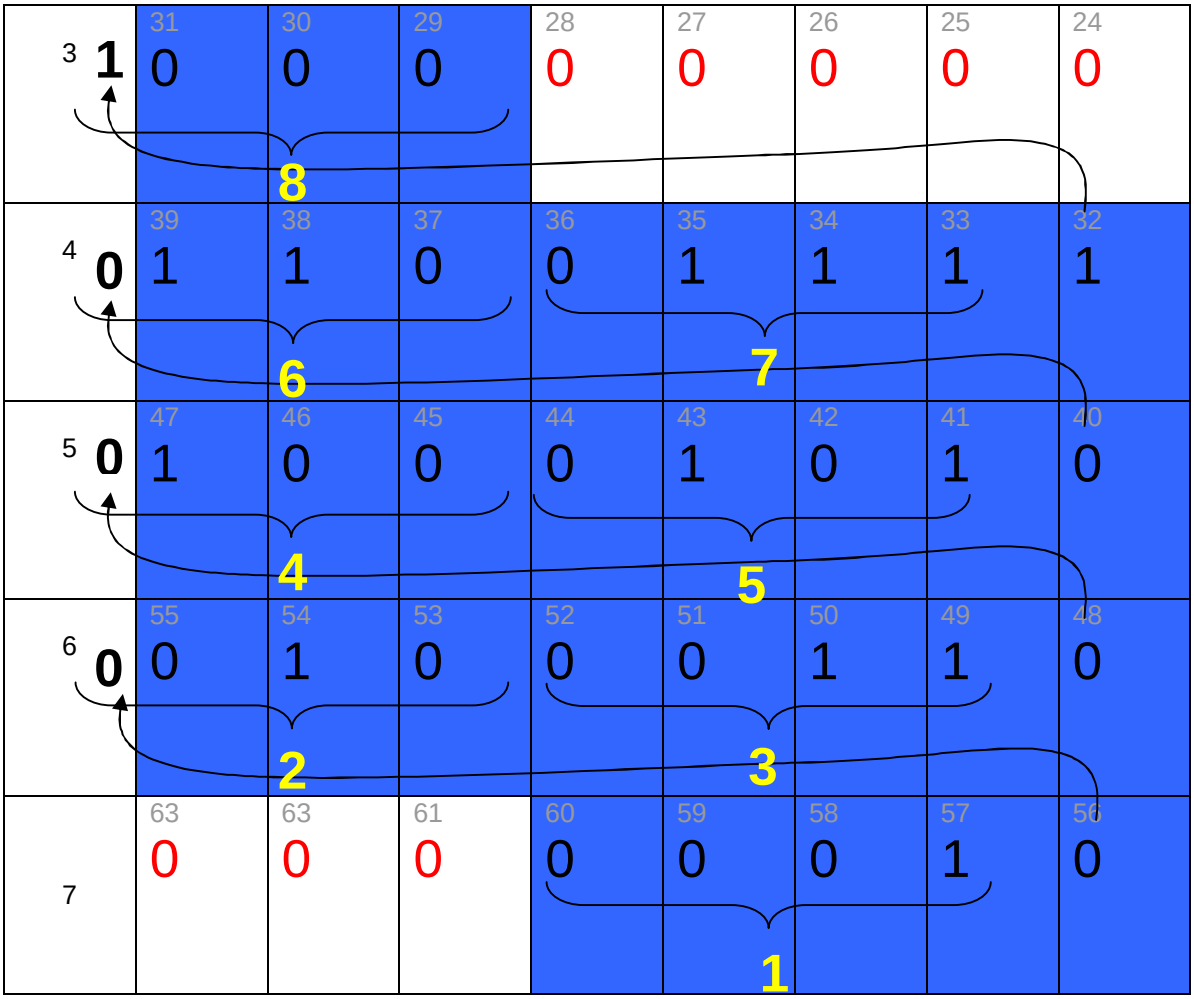

Figure 3-2 : Data recovery from little endian format

As an example for big endian format we have a 32 bit signal which also starts at bit 29 in a 8 byte CAN msg:

	7	6	5	4	3	2	1	0
0	7	6	5	4	3	2	1	0
1	15	14	13	12	11	10	9	8
2	23	22	21	20	19	18	17	16
3	31	30	29 msb	28	27	26	25	24
4	39	38	37	36	35	34	33	32
5	47	46	45	44	43	42	41	40
6	55	54	53	52	51	50	49	48
7	63	63 lsb	61	60	59	58	57	56

Figure 3-3 : Frame Layout big endian format

If we set the signal to 0x12345678 an send the CAN message with all other bits set to 0, we will receive a CAN message with 8 data bytes: 00 00 00 00 48 D1 59 E0.

If we look a the last 5 byte in bit-format:

Bytes	3		4		5		6		7	
Hex	0	0	4	8	D	1	5	9	E	0
Bits	0000	0000	0100	1000	1101	0001	0101	1001	1110	0000

Shift the bits 2 times to the right and you will get the signal value back:

Hex	0	1	2	3	4	5	6	7	8	0
Bits	0000	0001	0010	0011	0100	0101	0110	0111	1000	0000

3.3 Bus Layer

Description of all data structures, definitions and functions in the bus layer:

Data Structures:

- CANBUS_DATA CAN BUS OBJECT
- CANBUS_PARA BUS CONFIGURATION

Functions:

- CanBusInit() INITIALIZE CAN BUS MANAGEMENT
- CanBusIoCtl() CAN BUS I/O CONTROL
- CanBusRead() RECEIVE CAN FRAME
- CanBusWrite() SEND CAN FRAME
- CanBusEnable() ENABLE CAN BUS
- CanBusDisable() DISABLE CAN BUS
- CanBusTxHandler() CAN TX INTERRUPT HANDLER
- CanBusRxHandler() CAN RX INTERRUPT HANDLER
- CanBusNSHandler() CAN STATUS-CHANGE INTERRUPT HANDLER

3.3.1 CANBUS_DATA

Description:

This structure holds the runtime data for the CAN bus management.

Members:

```
typedef struct {
    CANBUS_PARA *Cfg;
    CPU_INT16S   Dev;
    CPU_INT16U   RxTimeout;
    CPU_INT16U   TxTimeout;
    CANFRM       BufTx[CANBUS_TX_QSIZE];
    CPU_INT16U   BufTxRd;
    CPU_INT16U   BufTxWr;
    CANFRM       BufRx[CANBUS_RX_QSIZE];
    CPU_INT16U   BufRxRd;
    CPU_INT16U   BufRxWr;
#if CANBUS_STAT_EN > 0
    CPU_INT16U   RxOkay;
    CPU_INT16U   TxOkay;
    CPU_INT16U   RxLost;
    CPU_INT16U   TxLost;
#endif
} CANBUS_DATA;
```

Member	Meaning
Cfg	The pointer to the read-only CAN bus configuration
Dev	This member holds the device ID, which is returned by the Open() function of the linked low level device driver.
RxTimeout	The timeout, which is used during the CanBusRead()
TxTimeout	The timeout, which is used during the CanBusWrite()
BufTx[]	The internal transmit queue which is used between CanBusWrite() and CanBusTxHandler() when the CAN bus is busy.
BufTxRd	The internal transmit queue read pointer.
BufTxWr	The internal transmit queue write pointer.
BufRx[]	The internal receive queue which is used between CanBusRxHandler() and CanBusRead().
BufRxRd	The internal receive queue read pointer.
BufRxWr	The internal receive queue write pointer.
RxOkay	This member holds the counter, which will be incremented every received CAN frame. This counter is incremented, when the CanBusRead() function successfully gives the CAN frame to the application layer.
TxOkay	The counter, which will be incremented every transmitted CAN frame. This counter is incremented, when the CAN transmit complete interrupt indicates, that the CAN frame is sent to the bus.
RxLost	This member holds the counter, which will be incremented for

	every received CAN frame which can not transferred to the application, due to a full receive queue.
`TxLost`	This member holds the counter, which will be incremented for every transmission CAN frame, which can not transferred to the transmit interrupt handler due to a full transmission queue

3.3.2 CANBUS_PARA

Description:

This structure contains the information for a bus. A bus represents one interface to the world.

Members:

```
typedef struct {
    CPU_BOOLEAN  Extended;
    CPU_INT32U   Baudrate;
    CPU_INT32U   BusNodeName;
    CPU_INT32U   DriverDevName;
    CPU_INT16S (*Init)  (CPU_INT32U);
    CPU_INT16S (*Open)  (CPU_INT16S, CPU_INT32U, CPU_INT16U);
    CPU_INT16S (*Close) (CPU_INT16S);
    CPU_INT16S (*IoCtl) (CPU_INT16S, CPU_INT16U, void *);
    CPU_INT16S (*Read)  (CPU_INT16S, CPU_INT08U *, CPU_INT16U);
    CPU_INT16S (*Write) (CPU_INT16S, CPU_INT08U *, CPU_INT16U);
    CPU_INT16U   Io[CAN_IO_FUNC_N];

} CANBUS_PARA;
```

Member	Meaning
`Extended`	The default configuration for the receive buffer
`Baudrate`	The baudrate in bit/s. If this is set to 0 then the CanBusEnable-function will not set the CAN device to active state.
`BusNodeName`	The bus node name, which must be used to open the interface with the can bus layer. This is a unique number.
`DriverDevName`	The driver device name, which must be used to open the interface with the low level device driver. This number is unique for a device driver.
`Init`	The function pointer to the CAN low level device driver XXX_Init() function (see chapter 4.1 “Driver Layer”)
`Open`	The function pointer to the CAN low level device driver XXX_Open() function (see chapter 4.1 “Driver Layer”)
`Close`	The function pointer to the CAN low level device driver XXX_Close() function (see chapter 4.1 “Driver Layer”)
`IoCtl`	The function pointer to the CAN low level device driver XXX_IoCtl() function (see chapter 4.1 “Driver Layer”)
`Read`	The function pointer to the CAN low level device driver XXX_Read() function (see chapter 4.1 “Driver Layer”)
`Write`	The function pointer to the CAN low level device driver XXX_Write() function (see chapter 4.1 “Driver Layer”)
`Io`	The map for all needed IO function codes. The corresponding function codes shall be provided by the CAN low level device driver (see chapter 4.1.4 “XXX_IoCtl”)

Additional Information:

Separation of bus node from driver device is necessary when different CAN modules are used, i.e. on a processor with integrated CAN module and also external CAN module(s).

Example: On a MPC565 processor TouCAN A, B, C are used and also 3 external SJA1000 CAN modules. Then 6 CAN nodes are available (range 0-5), but driver devices from TouCAN range 0-2 and driver devices from SJA1000 do also range from 0-2.
In this case a possible configuration could be:

Configuration TouCAN A Bus node 0 Driver device 0
Configuration TouCAN B Bus node 1 Driver device 1
Configuration SJA1000 2 Bus node 2 Driver device 1
Configuration SJA1000 3 Bus node 3 Driver device 2
Configuration TouCAN C Bus node 4 Driver device 2
Configuration SJA1000 1 Bus node 5 Driver device 0

If only one CAN device is used then bus node and driver device should be set to equal values.

3.3.3 CanBusInit()

Description:

Initializes the CAN bus management. This includes the creation of all needed OS-service components and the clearing of list of bus elements.

Prototype:

```
CPU_INT16S CanBusInit(CPU_INT32U arg);
```

Parameter	Meaning
arg	Not used

Note:

This function has the standard device driver interface, described in the porting chapter of the user manual. This allows the CAN bus handling via the standard device driver interface

Return Value:

One of the following error codes is returned: CAN_ERR_NONE, CAN_ERR_OSALLOC, CAN_ERR_OSQUEUE or CAN_ERR_OSSEM.

3.3.4 CanBusIoCtl()

Description:

This function performs a special action on the opened device. The function code func defines what the caller want to do.

Prototype:

```
CPU_INT16S CanBusIoCtl(CPU_INT16S  busId,
                       CPU_INT16U  func,
                       void       *argp);
```

Parameter	Meaning
busId	Unique bus identifier
func	The function code
argp	Pointer to argument, specific to the function code

Additional Information:

Function Code	Meaning
CANBUS_RESET	Reinitialize the CAN bus; all pending transmissions will be canceled and received messages will be lost
CANBUS_FLUSH_TX	Remove all entries out of the TX queue
CANBUS_FLUSH_RX	Remove all entries out of the RX queue
CANBUS_SET_TX_TIMEOUT	Set TX timeout value
CANBUS_SET_RX_TIMEOUT	Set RX timeout value

Return Value:

One of the following error codes is returned: CAN_ERR_NONE, CAN_ERR_BUSID, CAN_ERR_NULLPTR, CAN_ERR_OPEN, CAN_ERR_OSSEMPOST, CAN_ERR_OSFREE or CAN_ERR_IOCTRLFUNC.

3.3.5 CanBusRead()

Description:

This function is called by the application to obtain a frame from the opened CAN bus. The function will wait for a frame to be received on the CAN bus or until the configured timeout is reached.

Prototype:

```
CPU_INT16S CanBusRead(CPU_INT16S   busId,
                      void        *buffer,
                      CPU_INT16U   size);
```

Parameter	Meaning
busId	Unique bus identifier
buffer	Pointer to CAN frame of type CANFRM
size	Size of buffer in bytes (must be equal to sizeof(CANFRM))

Note:

A timeout is not handled as an error, therefore the return value in this case is 0.

Return Value:

One of the error codes CAN_ERR_BUSID, CAN_ERR_NULLPTR, CAN_ERR_FRMSIZE or CAN_ERR_OSFREE if an error is detected. Otherwise the number of bytes of a CAN frame is returned.

3.3.6 CanBusWrite()

Description:

This function is called by the application to send a CAN frame on the opened CAN bus. The function will wait for the buffer to empty out if the buffer is full. The function returns to the application if the buffer doesn't empty within the configured timeout. A timeout value of 0 (standard) means, that the calling function will wait forever for the buffer to empty out..

Prototype:

```
CPU_INT16S CanBusWrite(CPU_INT16S   busId,
                       void        *buffer,
                       CPU_INT16U   size);
```

Parameter	Meaning
busId	Unique bus identifier
buffer	Pointer to CAN frame of type CANFRM
size	Size of buffer in bytes (must be equal to sizeof(CANFRM))

Return Value:

One of the error codes CAN_ERR_BUSID, CAN_ERR_NULLPTR, CAN_ERR_FRMSIZE, CAN_ERR_OSSEMPEND, CAN_ERR_OSALLOC or CAN_ERR_OSQUEUE if an error is detected. Otherwise the number of bytes of a CAN frame is returned.

3.3.7 CanBusEnable()

Description:

This function checks, if a free CAN bus data element is available. If so, this element is initialized as set in the configuration struct of CANBUS_PARA. If a valid baudrate is configured then the CAN bus driver is used to initialize the CAN bus with the defined baud rate and set the CAN bus in active state. If a invalid baudrate or 0 is configured then this function does return before setting the bus to active state. Therefore the bus is ready for communication.

Prototype:

```
CPU_INT16S CanBusEnable(CANBUS_PARA *cfg);
```

Parameter	Meaning
cfg	Configuration of bus

Additional Information:

The configured CAN bus controller will be opened with the low level device driver and not closed at the end of the creation. This is done to ensure, that no other function/task tries to use the CAN controller. In other words: this implementation of the CAN bus management needs exclusive access to the CAN controller.

Return Value:

One of the following error codes is returned: CAN_ERR_NONE, CAN_ERR_NULLPTR, CAN_ERR_BUSID, CAN_ERR_OPEN or CAN_ERR_ENABLE.

3.3.8 CanBusDisable()

Description:

This function checks, if a CAN bus data element is in use. If so, the CAN bus will be closed with the low level device driver, e.g. the CAN device will be set to a bus off state.

Prototype:

```
CPU_INT16S CanBusDisable(CPU_INT16S busId);
```

Parameter	Meaning
busId	Unique bus identifier

Return Value:

One of the following error codes is returned: CAN_ERR_NONE, CAN_ERR_BUSID or CAN_ERR_NULLPTR.

3.3.9 CanBusTxHandler()

Description:

The calling interrupt vector has to place the bus identifier as parameter to the handler. The value of busId must be the BusNodeName.

Prototype:

```
void        CanBusTxHandler(CPU_INT16S busId);
```

Parameter	Meaning
busId	Unique bus identifier

3.3.10 CanBusRxHandler()

Description:

The calling interrupt vector has to place the bus identifier as parameter to the handler. The value of busId must be the BusNodeName.

Prototype:

```
void        CanBusRxHandler(CPU_INT16S busId);
```

Parameter	Meaning
busId	Unique bus identifier

Additional Information:

This function provides the hook-function CanBusRxHook(..). The hook-function can be en/disabled via configuration parameter in file can_cfg.h. The hook-function can be used to bypass µC/CAN-functionality for a received CAN frame.

3.3.11 CanBusNSHandler()

Description:

The calling interrupt vector has to place the bus identifier as parameter to the handler. The value of busId must be the BusNodeName.

Prototype:

```
void        CanBusNSHandler(CPU_INT16S busId);
```

Parameter	Meaning
busId	Unique bus identifier

Additional Information:

This function provides the hook-function CanBusNsHook(..). The hook-function can be en/disabled via configuration parameter in file can_cfg.h. The hook-function can be used to react on node status events, e.g. bus off.

3.4 Error Codes

This chapter contains the description of all error codes.

Error Code	The error code indicates, that ...
CAN_ERR_NONE	no error is detected.
CAN_ERR_NULLPTR	a parameter pointer is NULL
CAN_ERR_BUSID	the bus identifier is out of range
CAN_ERR_FRMSIZE	the buffer size (CAN frame size) is invalid
CAN_ERR_OPEN	the driver indicates, that the CAN bus can't be opened
CAN_ERR_ENABLE	the CAN bus can't be enabled
CAN_ERR_IOCTRLFUNC	the I/O function code is out of range
CAN_ERR_NULLMSG	the given message identifier is not found
CAN_ERR_MSGID	the message identifier is out of range
CAN_ERR_MSGUNUSED	unused signals are linked to the message
CAN_ERR_MSGCREATE	the message can't be created
CAN_ERR_SIGID	the signal identifier is out of range
CAN_ERR_NULLSIGCFG	the signal is not configured
CAN_ERR_CANSIZE	the given signal size is invalid, e.g. is not 1, 2 or 4
CAN_ERR_SIGCREATE	the signal can't be created
CAN_ERR_FRMWIDTH	the given frame width is not 1, 2 or 4
CAN_ERR_OSFREE	the CAN frame can't be released to the memory pool
CAN_ERR_OSQUEUE	the CAN frame can't be set into the RX or TX queue
CAN_ERR_OSALLOC	the CAN frame can't be allocated from the memory pool
CAN_ERR_OSSEM	the CAN semaphore(s) can't be created
CAN_ERR_OSQPEND	the CAN RX queue is empty and timeout error is detected
CAN_ERR_NOFRM	the CAN RX queue is empty
CAN_ERR_OSSEMPEND	the CAN semaphore pending timeout error is detected
CAN_ERR_OSSEMPOST	the CAN semaphore posting timeout error is detected
CAN_ERR_OSQACCEPT	the CAN TX queue is empty

Note: the global variable "`CPU_INT16S can_errnum`" holds the last detected error code.

4 Hardware Abstraction

The driver layer abstracts the different CAN controllers from the common usable CAN library. This chapter describes the interface of the low level device driver functions.

There are some parameters in function declarations, which are stated to be as 'unused'. This is done to get a common device driver interface (even beside the CAN device), similar to the Posix driver interface.

4.1 Driver Layer

The driver must contain the following functions:

- XXX_Init INITIALIZE CAN CONTROLLER
- XXX_Open LOCK A CAN CONTROLLER DEVICE
- XXX_Close RELEASE A CAN CONTROLLER DEVICE
- XXX_IoCtl CONTROL THE CAN CONTROLLER DEVICE
- XXX_Read READ DATA FROM A CAN CONTROLLER DEVICE
- XXX_Write WRITE DATA TO A CAN CONTROLLER DEVICE

Where XXX is the exact chip identification.

Examples:

A device driver for an external SJA1000 chip, the init function is labeled "SJA1000_Init"

A device driver for the internal CAN controllers #0..#2 of the TriCore 1796, the init function is labeled "TC1796_Init".
(Note: the three CAN controllers within the chip are addressed via the "device name". Details are described later in this chapter).

4.1.1 XXX_Init

Description:

This function must initialize the CAN controller selected by it's bus device name, clearing all message buffers (if available) and leave the CAN controller in bus off state.

Prototype:

```
void       XXX_Init(CPU_INT32U arg);
```

Parameter	Meaning
arg	bus device name

4.1.2 XXX_Open

Description:

This function marks the given CAN device as used, e.g. it locks the device. The return value is the identifier of the device and must be used for further actions with this device.

Prototype:

```
CPU_INT16S  XXX_Open(CPU_INT16S  drv,
                     CPU_INT32U  devName,
                     CPU_INT16U  mode);
```

Parameter	Meaning
drv	bus node name which must be used by the interrupt routine to access the can bus layer.
devName	device name which indicates the device within the controller
mode	not used

Return Value:

Device identifier for success or -1 if an error occurs.

Note:

Separation of bus node from driver device is necessary when different CAN modules are used, i.e. on a processor with integrated CAN module and also external CAN module(s).
Example: On a MPC565 processor TouCAN A, B, C are used and also 3 external SJA1000 CAN modules. Then 6 CAN nodes are available, but driver devices from TouCAN range 0-2 and driver devices from SJA1000 do also range from 0-2. To get unique device identification the bus node name is used. (see also chapter 5.1 Configure the CAN bus)

4.1.3 XXX_Close

Description:

This function releases the given CAN device as unused, e.g. it removes the device lock.

Prototype:

```
CPU_INT16S  XXX_Close(CPU_INT16S devId);
```

Parameter	Meaning
devId	device identifier, returned by XXX_Open()

Return Value:

Zero for success or -1 if an error occurs.

4.1.4 XXX_IoCtl

Description:

This function controls the given CAN device. The parameter func defines, which control operation the user wants to perform.

Prototype:

```
CPU_INT16S  XXX_IoCtl(CPU_INT16S  devId,
                      CPU_INT16U  func,
                      void       *argp);
```

Parameter	Meaning
devId	device identifier, returned by XXX_Open()
func	function code
argp	optional function argument

Additional Information:

Function Code IO_<DRV_NAME>_*	Meaning	used by µC/CAN
SET_BAUDRATE	set the bus baudrate	yes
RX_STANDARD	configure the CAN receiver to receive only CAN standard identifiers.	no
RX_EXTENDED	configure the CAN receiver to receive only CAN extended identifiers.	yes*
START	starts the CAN controller interface. Most common is to set the CAN controller in active mode	yes
STOP	stop the CAN controller interface. Most common is to set the CAN controller in passive mode	yes
GET_NODE_STATUS	get the node status from the CAN controller	no**
TX_READY	get status if CAN controller is ready to send new CAN frame	yes

* is used at startup, but some CAN devices will receive both extended or standard identifier if no mask is set
** needed only if CanBusNSHandler() is used.

Note: It's possible that some CAN devices do not support the function codes RX_STANDARD and RX_EXTENDED (e.g. SJA1000) or that this filter setting must be done via other function codes, e.g. for LPC2xxx CAN device drivers this must be set via SET_xxx_FILTER function codes. Please see the appropriate CanDriverManual if the function code is supported.

Return Value:

Zero for success or -1 if an error occurs.

4.1.5 XXX_Read

Description:

This function reads the last received CAN frame from the CAN controller. If there is no received CAN frame, the corresponding bytes will be 0.

Prototype:

```
CPU_INT16S XXX_Read(CPU_INT16S  devId,
                    CPU_INT08U *buffer,
                    CPU_INT16U  size);
```

Parameter	Meaning
devId	device identifier, returned by XXX_Open()
buffer	Pointer to CAN frame of type CANFRM
size	Size of buffer (must be equal to sizeof(CANFRM))

Return Value:

Number of bytes in CAN frame or -1 if an error was detected.

4.1.6 XXX_Write

Description:

This function writes a CAN frame into the CAN controller for transmission.

Prototype:

```
CPU_INT16S XXX_Write(CPU_INT16S  devId,
                     CPU_INT08U *buffer,
                     CPU_INT16U  size);
```

Parameter	Meaning
devId	device identifier, returned by XXX_Open()
buffer	Pointer to CAN frame of type CANFRM
size	Size of buffer in bytes (must be equal to sizeof(CANFRM))

Return Value:

Number of bytes written to buffer or -1 if an error was detected.

5 Example with µC/CAN

This chapter shows the usage of µC/CAN with some example sourcecode. Note, that this sourcecode needs several includes and environment settings (configuration in can_cfg.h). This is not considered within this chapter, because this chapter should show the intended use of the library.

5.1 Configure the CAN Bus

To configure a CAN bus interface, a global (optional: const) variable of the type CANBUS_PARA (defined in can_bus.h) must be allocated in a user file, e.g. can_cfg.c, and filled with the corresponding configuration values.

Example: The internal CAN bus controller #0 of the TriCore 1796 shall be used for a simple test protocol. The communication parameters are specified to:

- Standard CAN Identifiers (11 Bits)
- Baud rate shall be 250 kbit/s

Source code:

```
#include “can_bus.h”

/* CAN Bus Configuration */

const CANBUS_PARA CanCfg = {
    CAN_FALSE,                                      /*-1-*/
    250000L,                                        /*-2-*/
    0L,                                             /*-3-*/
    0L,                                             /*-4-*/
    TC1796_Init,   TC1796_Open,  TC1796_Close,      /*-5-*/
    TC1796_IoCtl,  TC1796_Read,  TC1796_Write,
    { 10,  11,  12,  13,  14,  15,  16}             /*-6-*/
};
```

Description:

1. Configure CAN bus layer to use standard CAN-Identifiers
2. Bus speed shall be 250000 bit/s
3. Bus node name for usage in can bus layer
4. Driver device of low level device drivers
5. Links to low level device driver functions
6. Trivial mapping of function codes, which are needed from the low level device driver

Additional Information:

1. The CAN bus configuration in the example is done as a global constant to safe RAM space and get a write protected configuration.

2. This variable must be declared global, because the configuration data will be read during initialization **and using** the CAN bus layer.

3. Separation of bus node from driver device is necessary when different CAN modules are used, i.e. on a processor with integrated CAN module and also external CAN module(s).

   Example: On a MPC565 processor TouCAN A, B, C are used and also 3 external SJA1000 CAN modules. Then 6 CAN nodes are available (range 0-5), but driver devices from TouCAN range 0-2 and driver devices from SJA1000 do also range from 0-2.
   In this case a possible configuration could be:

Configuration TouCAN A Bus node 0 Driver device 0
Configuration TouCAN B Bus node 1 Driver device 1
Configuration SJA1000 2 Bus node 2 Driver device 1
Configuration SJA1000 3 Bus node 3 Driver device 2
Configuration TouCAN C Bus node 4 Driver device 2
Configuration SJA1000 1 Bus node 5 Driver device 0

If only one CAN device is used then bus node and driver device can be set to equal values.

5.2 Enable the CAN Bus

Enabling the CAN bus interface means, that the CAN controller will go in "*active*" mode and is ready for communication.

Example: The CAN bus (we have configured in the chapter above) shall be enabled.

Source code:

```
#include "can_bus.h"

void Com_Start(void)
{
    CPU_INT16S err;                                    /*-1-*/

    CanBusInit(0L);                                    /*-2-*/

    err = CanBusEnable(&CanCfg);                       /*-3-*/
    if (err == CAN_ERR_NONE) {

        /* e.g. start RX and TX tasks */

    }
}
```

Description:

1. Local variable to hold the error code of CanBusEnable() function call.

2. Initialize the CAN bus management layer. This function call must be performed one time (and only one time) after reset of the system.

3. Enable the configured CAN bus. The configuration variable *CanCfg* is assumed to be the global (constant) variable as described in chapter 6.1

5.3 Send and Receive CAN Frames

Sending and receiving CAN frames can be implemented in different ways. The simplest way is to disable all upper layers and use the CAN frames directly. When using an operating system (e.g. µC/OS), the RX and TX communication could be performed within two tasks.

Example: The CAN frame reception shall be done within a single RX-Task. Any received CAN frame shall be send back with the fixed CAN Identifier: 0x100.

Source code:

```
#include “can_frm.h”
#include “can_bus.h”

void Echo_Task(void *argp)
{
    CANFRM frm;                                          /*-1-*/
    CPU_INT16U  timeout;

    /* Task initialisation */
    timeout = 0;
    CanBusIoCtl(0, CANBUS_SET_RX_TIMEOUT,
                (void *)&timeout );                      /*-2-*/

    /* Endless task loop */
    while (1) {

        CanBusRead(0, (void *)&frm, sizeof(CANFRM));     /*-3-*/

        frm.Identifier = 0x100L;                         /*-4-*/

        CanBusWrite(0, (void *)&frm, sizeof(CANFRM));    /*-5-*/
    }
}
```

Description:

1. Local variable to hold the received and echoed CAN frame.
2. Set the RX timeout to 0 to enable the blocking mode, e.g. the function CanBusRead() will wait for a CAN frame forever. Otherwise set a timeout and check returned error code of -3- if a timeout has occurred.
3. Wait for the next received CAN frame.
4. After reception, change the CAN-Identifier in received frame
5. Send the received CAN frame with changed CAN-Identifier

5.4 Defining CAN Signals and Messages

To get a higher level of abstraction, the CAN signals and messages can be used to abstract the information mappings to the bus communication objects.

Example: The information 'Nodestatus' and 'CPU-Load' shall be sent in a CAN frame called 'Status' with the following definition:

CAN-Identifier	= 0x150
DLC	= 2
Payload Byte 0	= Current Nodestatus
Payload Byte 1	= CPU-Load

The 'Nodestatus' shall be settable via the CAN frame called 'Command' with the following definition:

CAN-Identifier	= 0x140
DLC	= 1
Payload Byte 0	= New Nodestatus

If the information 'Nodestatus' is changed, the following actions shall be done:

New Nodestatus = 1: Start the task 'Load-Task'
New Nodestatus = 2: Stop the task 'Load-Task'

Source code (Part 1):

```
#include "can_sig.h"

/* Signal Definition */

enum {
    S_NODESTATUS = 0, S_CPULOAD, S_MAX
};

const CANSIG_PARA CanSig[CANSIG_N] = {
                                      /*--- S_NODESTATUS */
    { CANSIG_UNCHANGED,               /* Initial Status  */
      1,                              /* Width in Bytes  */
      0,                              /* Initial Value   */
       StatusChange },                /* Callback Func.  */
                                      /*------ S_CPULOAD */
    { CANSIG_UNCHANGED,               /* Initial Status  */
      1,                              /* Width in Bytes  */
      0,                              /* Initial Value   */
      0 }                             /* No Callback     */
};
```

Additional Information:

1. The CAN signals can be defined without any CAN communication knowledge. No information about sending or receiving the information is needed during this state.

2. The enumeration is used to simplify the message definition and ensure to get a consistent message to signal mapping.

3. The CAN signal configuration is done as a global constant to safe RAM space and get a write protected configuration.

4. This variable must be declared globally, because the configuration data will be read while using the CAN signal layer.

5. The callback function definition may be omitted by configuring the CANSIG_CALLBACK_EN parameter to 0.

Source code (Part 2):

```
#include “can_msg.h”

/* Message Definition */

enum {
    M_STATUS = 0, M_COMMAND, M_MAX
};

const CANMSG_PARA CanMsg[CANMSG_N] = {
                                    /*------- M_STATUS */
    { 0x150L,                       /* CAN-Identifier  */
      CANMSG_TX,                    /* Message Type    */
       2,                           /* DLC of Message  */
       2,                           /* No. of Links    */
       { { S_NODESTATUS,            /* Signal ID       */
         0 },                       /* Byte Position   */
       { S_CPULOAD,                 /* Signal ID       */
         1 } } },                   /* Byte Position   */
                                    /*------ M_COMMAND */
    { 0x140L,                       /* CAN-Identifier  */
      CANMSG_RX,                    /* Message Type    */
       1,                           /* DLC of Message  */
       1,                           /* No. of Links    */
       { { S_NODESTATUS,            /* Signal ID       */
         0 } } }                    /* Byte Position   */
};
```

Additional Information:

1. The CAN messages can be defined without any knowledge of the information generation and/or usage of the payload. Only the mapping of information parts to the payload must be known.

2. The enumeration is used to simplify the message handling.

3. The CAN message configuration is done as a global constant to safe RAM space and get a write protected configuration.

4. This variable must be declared globally, because the configuration data will be read while using the CAN message layer.

5.5 Application using the CAN Signals

To encapsulate the CAN protocol from the application, it is advised to use only the CAN signals within the application part. The following source code makes use of the definitions explained in chapter 5.4 “Defining CAN Signals and Messages”:

Source code (Application Startup):

```
void App_Startup(void)
{
    CPU_INT16S   err;
    CANSIG_PARA *s;
    CANMSG_PARA *m;

    CanSigInit(0L);                                     -1-
#if CANSIG_STATIC_CONFIG == 0
    s = CanSig;
    while (s < &CanSig[S_MAX]) {                        -2-
        err = CanSigCreate(s);
        if (err < 0) {

            /* failure handling */
        }
        s++;
    }
#endif

    CanMsgInit(0L);                                     -3-
    m = CanMsg;
    while (m < &CanMsg[M_MAX]) {                        -4-
        err = CanMsgCreate(m);
        if (err < 0) {

            /* failure handling */
        }
        m++;
    }

    Com_Start();                                        -5-

    /* start application task */
};
```

Description:

1. Initialize the CAN signal layer
2. If dynamic signal handling is enabled loop through all constant CAN signal parameter list and create the signals
3. Initialize the CAN message layer
4. Loop through all constant CAN message parameter list and create the messages
5. Start the communication bus as described in chapter 6.2 “Enable the CAN bus”

Source code (Application Task):

```
/* Application */

void App_Task(void)
{
    /* Task initialisation */

    /* Endless task loop */
    while (1) {

        /* perform your application actions */

        /* set CPU load, e.g. in µC/OS-II: */
        CanSigWrite(S_CPULOAD, &OSCPUUsage, 1);         -1-

        Send_Status();                                  -2-
    }
}

/* Signal Callback Function */

void StatusChange(void *Signal, CANSIG_VAL_T *NewVal,
                  CPU_INT32U CallbackId)                -3-
{
    if (CallbackId == CANSIG_CALLBACK_READ_ID) {        -4-
        if (*NewVal == 1) {                             -5-

            /* start Load-Task */

      } else {

            /* stop Load-Task */

      }
    }
}
```

Description:

1. Anywhere in your application the write access to the CAN signal is possible.

2. This function will send the status message to the CAN bus. The source code of the function is shown below.

3. This callback-function is included in the signal configuration parameter list for the signal ‘status’. The function is called, when the change of the signal value is detected.

4. The parameter ‘CallbackId’ is used to identify where this callback-function was called from.

5. The parameter ‘NewVal’ holds the new CAN signal value and is used to determine the required start / stop action of the Load-Task.

5.6 Protocol using the CAN Messages

The communication protocol uses in most cases only the CAN messages. The application is responsible for the values of the mapped CAN signals. The following source code completes the example, started in chapter 6.4 “Defining CAN Signals and Messages”:

Source code (Protocol RX-Task):

This task could be started within Com_Start() to enable the receive activities as soon as the CAN bus is active.

```
void Rx_Task(void *argp)
{
    CANFRM     frm;                                          /*-1-*/
    CPU_INT16S msg;                                          /*-2-*/

    /* Task initialisation */
    CanBusIoCtl(0, CANBUS_SET_RX_TIMEOUT, 0);                /*-3-*/

    /* Endless task loop */
    while (1) {

        CanBusRead(0,(void *)&frm,sizeof(CANFRM));           /*-4-*/

        msg = CanMsgOpen(0, frm.Identifier, 0);              /*-5-*/
        if (msg >= 0) {

          CanMsgWrite(msg,(void *)&frm,sizeof(CANFRM));      /*-6-*/
        }
    }
}
```

Description:

1. Local variable to hold the received CAN frame.
2. Local variable to hold the handle to message of received CAN identifier
3. Set the RX timeout to 0 to enable the blocking mode, e.g. the function CanBusRead() will wait forever for a CAN frame.
4. Wait for the next received CAN frame on CAN node 0.
5. After reception, open the message with the received identifier.
6. On success, write the received CAN frame to this message. This results in decomposing the CAN frame to all linked CAN signals.

Source code (Protocol Transmit Status):

This function can be called in your transmission task to transmit a specific CAN message, or called within a callback function to transmit information in response to a received CAN message.

```
void Send_Status(void)
{
    CANFRM     frm;                                          /*-1-*/
    CPU_INT16S msg;                                          /*-2-*/

    msg = CanMsgOpen(0, 0x150L, 0);                          /*-3-*/
    if (msg >= 0) {

        CanMsgRead(msg, (void *)&frm, sizeof(CANFRM));       /*-4-*/

        CanBusWrite(0, (void *)&frm, sizeof(CANFRM));        /*-5-*/
    }
};
```

Description:

1. Local variable to hold the CAN frame for transmission.

2. Local variable to hold the handle to message which shall be transmitted

3. Open the CAN message with the CAN-Identifier 0x150.

4. On success, read the message to the local CAN frame. This results in collecting all linked signals and writing all information to the CAN frame at the defined locations.

5. Send this constructed CAN frame via the CAN bus 0.

References

µC/OS-II, The Real-Time Kernel, 2nd Edition
Jean J. Labrosse
R&D Technical Books, 2002
ISBN 1-57820-103-9

Embedded Systems Building Blocks
Jean J. Labrosse
R&D Technical Books, 2000
ISBN 0-87930-604-1

Contacts

Micriμm
949 Crestview Circle
Weston, FL 33327
USA
954-217-2036
954-217-2037 (FAX)
e-mail: sales@Micrium.com
WEB: www.Micrium.com

Embedded Office
August-Braun-Sraße 1
88239 Wangen
Germany
+49 7522 970 008-0
+49 7522 970 008-99 (FAX)
e-mail: info@embedded-office.de
WEB: www.embedded-office.de

